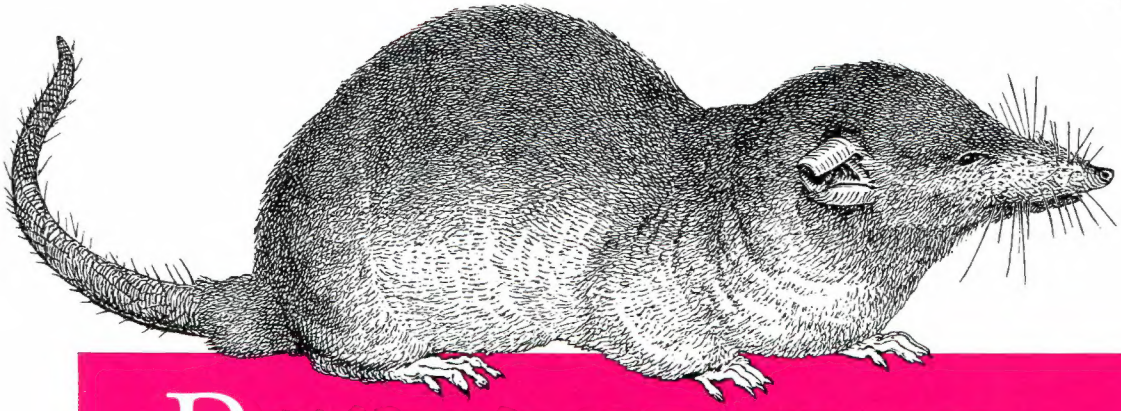
Подробные решения
на восьми языках программирования

2-е издание
обновленное и дополненное



Регулярные выражения

Сборник рецептов

СИМВОЛ®
O'REILLY®

Ян Гойвертс,
Стивен Левитан

Regular Expressions Cookbook

Second Edition

Jan Goyvaerts, Steven Levithan

O'REILLY®

Регулярные выражения
Сборник рецептов

Второе издание

Ян Гойвертс, Стивен Левитан

Санкт-Петербург — Москва
2015

Ян Гойвертс, Стивен Левитан

Регулярные выражения. Сборник рецептов, 2-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*Ю. Бочина*
Корректор	*Т. Иванкова*
Верстка	*Д. Орлова*

Гойвертс Я., Левитан С.

Регулярные выражения. Сборник рецептов, 2-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2015. – 704 с., ил.

ISBN 978-5-93286-221-6

Книга ведущих экспертов по регулярным выражениям содержит все необходимое для решения широкого спектра практических задач в виде более 140 проверенных рецептов. С ее помощью новички смогут обрести базовые навыки и освоить основные инструменты, а программисты и опытные пользователи найдут в ней множество важных мелочей. Каждый рецепт содержит шаблоны, пригодные для немедленного применения.

Второе обновленное издание охватывает диалекты регулярных выражений, используемые в восьми языках программирования: C#, Java, JavaScript, Perl, PHP, Python, Ruby и VB.NET. Вы познакомитесь с новыми приемами, научитесь обходить ловушки, имеющиеся в разных диалектах, и благодаря огромной библиотеке готовых решений сможете сэкономить свое время.

В книге представлены пошаговые решения наиболее часто встречающихся задач: проверка и форматирование ввода пользователя, поиск и обработка слов, спецсимволов и строк текста, работа с числами в различных форматах представления, работа с адресами URL, путями в файловой системе и IP-адресами, приемы работы с разметкой HTML, XML и с форматами обмена данными. Освоив материал книги, вы сможете полнее использовать все те возможности, которые предоставляет умелое применение регулярных выражений.

ISBN 978-5-93286-221-6
ISBN 978-1-449-31943-4 (англ)

© Издательство Символ-Плюс, 2015

Authorized Russian translation of the English edition of Regular Expressions Cookbook, Second Edition ISBN 9781449319434 © 2012 O'Reilly Media, Inc. All rights reserved. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 25.12.2014. Формат 70×100 $^{1}/_{16}$.
Печать офсетная. Объем 44 печ. л.

Оглавление

Предисловие 9

1. Введение в регулярные выражения 16

Определение регулярных выражений 16
Поиск с заменой с помощью регулярных выражений 22
Инструменты для работы с регулярными выражениями 25

2. Основные навыки владения регулярными выражениями 49

2.1. Соответствие литеральному тексту 50
2.2. Соответствие непечатным символам 53
2.3. Сопоставление с одним символом из нескольких 56
2.4. Сопоставление с любым символом 62
2.5. Сопоставление в начале и/или в конце строки 65
2.6. Сопоставление с целыми словами 70
2.7. Кодовые пункты Юникода, категории, блоки и алфавиты 74
2.8. Сопоставление с одной из нескольких альтернатив 90
2.9. Группы и сохранение части совпадения 92
2.10. Повторный поиск соответствия с ранее совпавшим текстом 96
2.11. Сохранение и именованные части совпадения 98
2.12. Повторение части регулярного выражения определенное число раз 103
2.13. Выбор минимального или максимального числа повторений 107
2.14. Устранение бесполезных возвратов 111
2.15. Предотвращение бесконтрольных повторений 114
2.16. Проверка соответствия без включения его в общее соответствие 118
2.17. Совпадение с одной из двух альтернатив по условию 127
2.18. Добавление комментариев в регулярные выражения 130
2.19. Вставка текстового литерала в замещающий текст 133
2.20. Вставка совпадения с регулярным выражением в замещающий текст . . . 136
2.21. Вставка части совпадения с регулярным выражением в замещающий текст 138
2.22. Вставка контекста совпадения в замещающий текст 142

3. Программирование с применением регулярных выражений 144

Языки программирования и диалекты регулярных выражений 144
3.1. Литералы регулярных выражений в исходных текстах 152
3.2. Импортирование библиотеки регулярных выражений 160

3.3. Создание объектов регулярных выражений 162
3.4. Установка параметров регулярных выражений 170
3.5. Проверка возможности совпадения в пределах испытуемой строки 179
3.6. Проверка совпадения со всей испытуемой строкой 186
3.7. Извлечение текста совпадения 192
3.8. Определение позиции и длины совпадения 199
3.9. Извлечение части совпавшего текста 205
3.10. Извлечение списка всех совпадений 213
3.11. Обход всех совпадений в цикле 219
3.12. Проверка полученных совпадений в программном коде 227
3.13. Поиск совпадения внутри другого совпадения 231
3.14. Замена всех совпадений 237
3.15. Замена совпадений с повторным использованием частей совпадений 245
3.16. Замена совпадений фрагментами, сгенерированными в программном коде 251
3.17. Замена всех совпадений внутри совпадений с другим регулярным выражением 258
3.18. Замена всех совпадений между совпадениями с другим регулярным выражением 260
3.19. Разбиение строки 266
3.20. Разбиение строки, сохранение совпадений с регулярным выражением . . 275
3.21. Построчный поиск 280
3.22. Конструирование парсера 284

4. Проверка и форматирование 300

4.1. Проверка адресов электронной почты 300
4.2. Проверка и форматирование телефонных номеров 307
4.3. Проверка международных телефонных номеров 314
4.4. Проверка дат в традиционных форматах 316
4.5. Точная проверка дат в традиционных форматах 321
4.6. Проверка времени в традиционных форматах 327
4.7. Проверка даты и времени в формате ISO 8601 330
4.8. Ограничение возможности ввода алфавитно-цифровыми символами 338
4.9. Ограничение длины текста 342
4.10. Ограничение числа строк в тексте 348
4.11. Проверка утвердительных ответов 354
4.12. Проверка номеров социального страхования 356
4.13. Проверка номеров ISBN 358
4.14. Проверка почтовых индексов 367
4.15. Проверка почтовых индексов, используемых в Канаде 368
4.16. Проверка почтовых индексов, используемых в Великобритании 369
4.17. Поиск адресов, содержащих номер почтового ящика 371
4.18. Преобразование из формата «имя фамилия» в формат «фамилия, имя» 373
4.19. Проверка сложности пароля 377
4.20. Проверка номеров кредитных карт 387
4.21. Европейские регистрационные номера плательщиков НДС 393

5. Слова, строки и специальные символы 400
5.1. Поиск определенного слова 400
5.2. Поиск любого слова из множества 403
5.3. Поиск похожих слов 406
5.4. Поиск любых слов, за исключением указанного 411
5.5. Поиск любого слова, за которым не следует указанное слово 413
5.6. Поиск любого слова, которому не предшествует определенное слово 415
5.7. Поиск близко расположенных слов 421
5.8. Поиск повторяющихся слов 429
5.9. Удаление повторяющихся строк 432
5.10. Совпадение с полными строками, содержащими определенное слово 438
5.11. Совпадение с полными строками, не содержащими определенное слово . . 441
5.12. Удаление ведущих и завершающих пробельных символов 442
5.13. Замена повторяющихся пробельных символов единственным пробелом . 446
5.14. Экранирование метасимволов регулярных выражений 448

6. Числа 454
6.1. Целые числа 454
6.2. Шестнадцатеричные числа 458
6.3. Двоичные числа 462
6.4. Восьмеричные числа 463
6.5. Десятичные числа 464
6.6. Удаление ведущих нулей 465
6.7. Числа в определенном диапазоне 467
6.8. Шестнадцатеричные числа в определенном диапазоне 474
6.9. Целые числа с разделителями 477
6.10. Вещественные числа 479
6.11. Числа с разделителями групп разрядов 483
6.12. Добавление в числа разделителей групп разрядов 484
6.13. Римские числа 490

7. Исходный код и файлы журналов 494
7.1. Ключевые слова 494
7.2. Идентификаторы 498
7.3. Числовые константы 498
7.4. Операторы 500
7.5. Однострочные комментарии 502
7.6. Многострочные комментарии 502
7.7. Все комментарии 504
7.8. Строки 505
7.9. Строки с экранированными символами 509
7.10. Литералы регулярных выражений 511
7.11. Встроенные документы 514
7.12. Обобщенный формат файла журнала 516
7.13. Комбинированный формат файлов журналов 520
7.14. Сообщения о недействительных ссылках в файле журнала веб-сервера . . 521

8. URL, пути и адреса в Интернете 525
8.1. Проверка адресов URL 525
8.2. Поиск адресов URL в тексте 529
8.3. Поиск в тексте адресов URL, заключенных в кавычки 532
8.4. Поиск в тексте адресов URL, заключенных в скобки 533
8.5. Преобразование адресов URL в ссылки 536
8.6. Проверка строк URN 537
8.7. Проверка универсальных адресов URL 540
8.8. Извлечение схемы из адреса URL 546
8.9. Извлечение имени пользователя из URL 548
8.10. Извлечение имени хоста из URL 551
8.11. Извлечение номера порта из URL 553
8.12. Извлечение пути из адреса URL 556
8.13. Извлечение строки запроса из URL 560
8.14. Извлечение фрагмента из URL 561
8.15. Проверка доменных имен 562
8.16. Сопоставление с адресами IPv4 565
8.17. Сопоставление с адресами IPv6 569
8.18. Проверка путей в Windows 585
8.19. Выделение элементов путей в Windows 589
8.20. Извлечение буквы устройства из путей в Windows 594
8.21. Извлечение имен сервера и разделяемого ресурса из пути в формате UNC 595
8.22. Извлечение имен папок из путей в Windows 596
8.23. Извлечение имени файла из пути Windows 599
8.24. Извлечение расширения имени файла из пути Windows 600
8.25. Удаление недопустимых символов из имен файлов 601

9. Разметка и форматы данных 603
Обработка разметки и данных в различных форматах с помощью регулярных выражений 603
9.1. Поиск тегов XML 612
9.2. Заменить тег <b> тегом 631
9.3. Удаление всех XML-подобных тегов, за исключением и . . . 635
9.4. Сопоставление с именами XML 639
9.5. Преобразование простого текста в HTML добавлением тегов <p> и
 . . 646
9.6. Декодирование мнемоник XML 650
9.7. Поиск определенных атрибутов в XML-подобных тегах 653
9.8. Добавление атрибута cellspacing в теги <table>, где этот атрибут отсутствует 658
9.9. Удаление XML-подобных комментариев 662
9.10. Поиск слов в XML-подобных комментариях 667
9.11. Изменение разделителя, используемого в файлах CSV 672
9.12. Извлечение полей CSV из определенного столбца 677
9.13. Сопоставление с заголовком раздела в файле INI 681
9.14. Сопоставление с разделом в файле INI 683
9.15. Сопоставление с парами имя-значение в файле INI 684

Алфавитный указатель 687

Предисловие

В течение последнего десятилетия наблюдался неуклонный рост интереса к регулярным выражениям. На сегодняшний день все популярные языки программирования включают мощные библиотеки поддержки этого инструмента или даже имеют реализацию этой поддержки, встроенную непосредственно в сам язык. Многие разработчики, опираясь на поддержку регулярных выражений, предоставляют пользователям своих приложений возможность поиска или фильтрации данных с их помощью. Регулярные выражения присутствуют практически везде.

Издано множество книг, помогающих освоить регулярные выражения. Большинство из них прекрасно справляются с задачей описания их синтаксиса, дополнительно предоставляя некоторые примеры и справочную информацию. Но среди них нет такой, которая демонстрировала бы основанные на использовании регулярных выражений решения для широкого круга практических задач, связанных с обработкой текста, возникающих в разнообразных интернет-приложениях. Мы, Стив и Ян, решили данной книгой восполнить этот пробел.

Мы ставили своей целью показать, как можно применять регулярные выражения в ситуациях, в которых те, кто имеет недостаточный опыт работы с этим инструментом, могли бы решить, что их применить невозможно, а искушенные программисты могли бы посчитать, что для решения подобных задач не стоит их использовать. Поскольку в настоящее время регулярные выражения получили повсеместную поддержку, они часто оказываются удобным инструментом, который может применяться конечными пользователями даже без посредничества программистов. Да и сами программисты могут сэкономить время, используя регулярные выражения для решения задач извлечения и изменения информации, не тратя сил и времени на создание соответствующего процедурного программного кода или изучение описаний возможностей различных вспомогательных библиотек.

Проблемы, связанные с многообразием версий

Как и для любых других инструментов, популярных в мире информационных технологий, существует множество различных реализаций

поддержки регулярных выражений, обладающих различной степенью совместимости. В результате это привело к появлению множества различных *диалектов* регулярных выражений, которые не всегда одинаково интерпретируют одно и то же регулярное выражение, если вообще способны его интерпретировать.

Во многих книгах отмечается наличие диалектов и указываются некоторые из их отличий. Но нередко какие-то диалекты в определенных ситуациях вообще не упоминаются, особенно если в этих диалектах отсутствуют какие-то возможности, тогда как можно было бы показать альтернативное решение или обходной путь. Это становится препятствием, когда приходится работать с разными диалектами регулярных выражений в различных приложениях или языках программирования.

Отдельные заявления, встречающиеся в литературе, такие как «в настоящее время все используют регулярные выражения в стиле языка Perl», к сожалению, выводят из рассмотрения широкий круг несовместимостей. Даже между пакетами реализации регулярных выражений «в стиле языка Perl» имеются существенные различия, к тому же сам язык Perl непрерывно развивается дальше. Чрезмерно упрощенное освещение проблемы может приводить к тому, что программист будет проводить часы бесплодных поисков в отладчике, вместо того чтобы исследовать особенности используемой реализации регулярных выражений. И даже когда он обнаружит, что та или иная особенность отсутствует в реализации, он не будет знать, как обойти этот недостаток.

Данная книга является первой, где на всем протяжении наиболее распространенные и обладающие богатыми возможностями диалекты регулярных выражений рассматриваются параллельно.

Для кого написана эта книга

Издание предназначено для тех, кто постоянно работает с текстовой информацией на компьютере, то есть обрабатывает огромные массивы документов, работает с текстом в текстовых редакторах или разрабатывает программное обеспечение, реализующее анализ и управление текстовой информацией. Регулярные выражения являются превосходным инструментом для решения таких задач. Эта книга содержит все, что необходимо знать об этом инструменте. От читателя не требуется наличие какого-либо опыта работы с регулярными выражениями, потому что в книге объясняются даже самые основные их положения.

Если вы уже знакомы с регулярными выражениями, то найдете здесь множество подробностей, о которых часто умалчивают другие книги или электронные издания. Если вам уже приходилось сталкиваться с ситуацией, когда одно и то же регулярное выражение работает в одном приложении, но не работает в другом, вы найдете весьма ценным подробное описание семи диалектов наиболее распространенных регулярных выражений, представленное в этой книге. Мы организовали книгу как сборник рецептов, поэтому вы можете начинать чтение непосредственно

с интересующей вас темы. А если вы прочтете это издание от корки до корки, то станете мастером регулярных выражений мирового уровня.

Эта книга даст вам все необходимые сведения о регулярных выражениях, даже если вы не занимаетесь программированием. Если вам требуется использовать регулярные выражения в текстовом редакторе, в инструменте поиска или в каком-либо другом приложении, где имеется поле ввода с надписью «regex» (регулярное выражение), вы можете приступать к чтению этой книги, даже не обладая опытом программирования. В большинстве рецептов, которые приводятся в книге, даны решения, основанные исключительно на применении одного или нескольких регулярных выражений.

Для программистов в главе 3 приводятся полные сведения, необходимые для реализации регулярных выражений в исходном программном коде. Данная глава предполагает, что читатель обладает знаниями об основных особенностях выбранного им языка программирования, но не требует наличия опыта использования регулярных выражений в исходном программном коде.

Охватываемые технологии

.NET, Java, JavaScript, PCRE, Perl, Python и Ruby – это не просто броские названия. Они представляют семь диалектов регулярных выражений, рассматриваемых в данной книге. Мы в равной степени уделим внимание всем семи диалектам. В особенности мы позаботимся о том, чтобы указать все несовместимости, существующие между ними, которые нам удалось обнаружить.

Глава, имеющая отношение к программированию (глава 3), содержит листинги исходных текстов на языках C#, Java, JavaScript, PHP, Perl, Python, Ruby и VB.NET. Здесь также для каждого рецепта приводятся решения и их описания для всех восьми языков. Поскольку это приводит к тому, что в главе многое повторяется не один раз, вы легко можете пропускать обсуждения, связанные с языками программирования, которые вас не интересуют, – достаточно следить за тем, что относится к выбранному вами языку.

Организация книги

В первых трех главах описываются наиболее ценные инструменты и даются основные сведения, позволяющие начать использование регулярных выражений. Каждая последующая глава представляет различные регулярные выражения, подробно исследуя какой-то один круг задач обработки текста.

Глава 1 «Введение в регулярные выражения» объясняет назначение регулярных выражений и знакомит с набором инструментов, способных облегчить их изучение, создание и отладку.

Глава 2 «Основные навыки владения регулярными выражениями» описывает каждый элемент и каждую особенность регулярных выражений, а также содержит рекомендации по их эффективному использованию. Она представляет собой полный учебник по регулярным выражениям.

Глава 3 «Программирование с применением регулярных выражений» демонстрирует приемы программирования и содержит листинги с исходными текстами на всех языках программирования, описываемых в этой книге, где используются регулярные выражения.

Глава 4 «Проверка и форматирование» содержит рецепты обработки информации, вводимой пользователем, такой как даты, номера телефонов и почтовые индексы, используемые в разных странах.

В главе 5 «Слова, строки и специальные символы» рассматриваются наиболее типичные задачи обработки текста, такие как проверка наличия или отсутствия в строках определенных слов.

В главе 6 «Числа» показывается, как определять целые и вещественные числа при вводе в различных форматах.

Глава 7 «Исходный код и файлы журналов» представляет строительные блоки для синтаксического анализа исходного программного кода и других текстовых документов, а также показывает, как с помощью регулярных выражений обрабатывать файлы журналов.

Глава 8 «URL, пути и адреса в Интернете» демонстрирует, как анализировать и манипулировать строками, часто используемыми в Интернете и в системе Windows в процессе поиска.

Глава 9 «Разметка и обмен данными» охватывает приемы манипулирования текстом в форматах HTML, XML, CSV (comma-separated values – данные, разделенные запятыми) и конфигурационными файлами в формате INI.

Типографские соглашения

В этой книге используются следующие типографские соглашения:

Курсив

Курсивом выделяются новые термины, адреса URL, адреса электронной почты, имена и расширения файлов.

Моноширинный шрифт

Используется для представления листингов программ, элементов программного кода, таких как имена переменных или функций, значений, возвращаемых в результате применения регулярных выражений, а также обрабатываемых элементов или текста, к которым применяется регулярное выражение. Это может быть содержимое строковой переменной, поля ввода в приложении или файла на диске.

Моноширинный курсивный шрифт

Используется для представления текста, который должен замещаться значениями, вводимыми пользователем, или для представления значений, определяемых контекстом.

‹Регулярное»выражение›

Так представляются регулярные выражения сами по себе или в том виде, в каком они вводятся в поле ввода в приложениях. Пробелы внутри регулярных выражений обозначаются серыми кружочками, чтобы сделать их более заметными. Пробелы не выделяются серыми кружками в режиме свободного форматирования (free-spacing mode), в котором пробелы игнорируются.

«Замещаемый»текст»

Так будет представлен текст, соответствующий регулярному выражению, который будет замещен при выполнении операции поиска с заменой. Пробелы в замещаемом тексте обозначаются серыми кружочками, чтобы сделать их более заметными.

Совпавший текст

Так будет представлен фрагмент текста, соответствующий регулярному выражению.

Многоточие внутри регулярных выражений указывает место, которое должно быть заполнено вами, прежде чем вы сможете воспользоваться ими. Сопутствующий текст поясняет, что должно быть подставлено вместо многоточия.

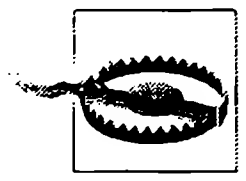
CR, LF и CRLF

Обозначения CR, LF и CRLF в рамках представляют фактические разрывы текстовых строк, а не специальные экранированные последовательности, такие как \r, \n и \r\n. Такие строки могут создаваться в многострочных полях ввода нажатием клавиши Enter, или при использовании многострочных строковых констант в исходных текстах программ, таких как строковые литералы в языке C#, или в строках, окруженных тройными кавычками, в языке Python.

↵

Эта стрелка, имеющаяся на клавише Return или Enter, говорит о том, что текст был перенесен исключительно для того, чтобы уместить его по ширине книжной страницы. При вводе такого текста не нужно нажимать клавишу Enter – просто продолжайте ввод в той же самой строке.

Этот значок отмечает советы, предложения и примечания общего характера.



Этот значок отмечает предупреждения и предостережения.

Использование программного кода примеров

Данная книга призвана помочь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить значительные отрывки программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы цитируете данную книгу или примеры из нее при ответах на вопросы, получение разрешения не требуется. При включении в вашу документацию существенных объемов программного кода примеров из этой книги вам необходимо получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Regular Expressions Cookbook by Jan Goyvaerts and Steven Levithan. Copyright 2012 Jan Goyvaerts and Steven Levithan, 978-1-449-31943-4».

За разрешением на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) – это виртуальная библиотека, содержащая авторитетную информацию в виде книг и видеоматериалов, созданных ведущими специалистами в области технологий и бизнеса.

Профессионалы в области технологий, разработчики программного обеспечения, веб-дизайнеры, а также бизнесмены и творческие работники используют Safari Books Online как основной источник информации для проведения исследований, решения проблем, обучения и подготовки к сертификационным испытаниям.

Библиотека Safari Books Online предлагает широкий выбор продуктов и тарифов для организаций, правительственных учреждений и физических лиц. Подписчики имеют доступ к поисковой базе данных, содержащей информацию о тысячах книг, видеоматериалов и рукописей от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesly Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal

Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology и десятков других. За подробной информацией о Safari Books Online обращайтесь по адресу *http://www.safaribooksonline.com/*.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на веб-странице, посвященной этой книге:

http://oreilly.com/catalog/9781449319434

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о наших книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на сайте *http://www.oreilly.com*.

Ищите нас на Facebook: *http://facebook.com/oreilly*.

Следуйте за нами в Твиттере: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Мы выражаем свою признательность Энди Ораму (Andy Oram), нашему редактору из издательства O'Reilly Media, Inc., за сопровождение этого проекта от начала и до конца. Мы также благодарим Джеффри Фридла (Jeffrey Friedl), Зака Грента (Zak Greant), Николая Линдберга (Nikolaj Lindberg) и Яна Морса (Ian Morse) за техническое рецензирование первого издания и Николая Линдберга, Джудит Майерсон (Judith Myerson) и Зака Грента за рецензирование второго издания, что позволило сделать эту книгу более последовательной и точной.

1

Введение в регулярные выражения

Раз уж вы открыли эту книгу, то наверняка захотите ввести в свой программный код какие-то из несуразных строк, состоящих из круглых скобок и вопросительных знаков, которые вам встретятся в этих главах. Если вы готовы включиться в игру, будьте нашим гостем: реальные регулярные выражения и их описания вы найдете в главах с 4 по 9.

Однако и начальные главы помогут вам сэкономить немало времени в будущем. Например, в этой главе будут представлены некоторые утилиты (часть из них создана авторами книги, Яном и Стивеном), которые позволяют проверять и отлаживать регулярные выражения перед включением их в программный код, где отыскать возможные ошибки будет намного труднее. Кроме того, начальные главы покажут вам, как использовать различные особенности и возможности регулярных выражений, способные облегчить вашу жизнь; дадут понимание регулярных выражений с точки зрения их оптимизации; покажут тонкие отличия в том, как обрабатываются регулярные выражения в разных языках программирования и даже в различных версиях одного и того же языка.

Мы приложили немалые усилия при подготовке этих базовых сведений в надежде, что вы начнете изучение с них или, если вы уже столкнулись с проблемами, связанными с использованием регулярных выражений, то приняли решение со всем этим разобраться.

Определение регулярных выражений

В контексте этой книги под *регулярными выражениями* понимаются определенные виды текстовых шаблонов, которые могут использоваться во многих современных приложениях и языках программирования. Вы можете использовать их, чтобы убедиться, что введенный текст вписывается в заданный шаблон; отыскивать внутри больших объемов текста фрагменты, соответствующие шаблону; замещать текст, соответст-

вующий шаблону, другим текстом или переупорядочивать фрагменты совпавшего текста; разбивать блоки текста на списки более мелких фрагментов или даже делать не столь разумные вещи. Эта книга поможет вам точно понимать, что вы делаете, и избегать ошибок.

История появления термина «регулярное выражение»

Термин «регулярное выражение» пришел из математики и теоретической информатики, где он отражает свойство математических выражений, называемое *регулярностью*. Такие выражения могут быть реализованы программно с использованием детерминированных конечных автоматов (ДКА). ДКА – это механизм с конечным числом состояний, не поддерживающий возможность возврата (backtracking).

Текстовые шаблоны, использовавшиеся в самых ранних версиях инструментов *grep*, были регулярными выражениями в математическом смысле. Несмотря на то что имя прижилось, современные регулярные выражения в стиле языка Perl фактически не являются регулярными выражениями в математическом смысле. Они реализованы с применением механизма недетерминированных конечных автоматов (НКА). Вы вскоре поближе познакомитесь с понятием возврата. Программистам-практикам достаточно усвоить из этого примечания следующее: высоколобым компьютерным теоретикам приходится мириться с тем, что строгие теоретические модели заменяются технологиями, более полезными в реальном мире.

При грамотном использовании регулярные выражения способны упростить задачи программирования и обработки текстов, а также обеспечить решение многих задач, которые вообще не могли бы быть решены без использования регулярных выражений. Вам могут потребоваться десятки, если не сотни, строк процедурного программного кода, чтобы извлечь все адреса электронной почты из документа, и такой код будет очень сложно написать и сопровождать. Однако с применением регулярных выражений, как показано в рецепте 4.1, решение этой задачи может быть реализовано всего лишь парой строк кода, а возможно, даже и одной строкой.

Но если попытаться взвалить слишком много на одно-единственное регулярное выражение или задействовать регулярные выражения там, где они малопригодны, вы поймете, почему некоторые говорят:[1]

---

[1] Историю этой цитаты Джеффри Фридл прослеживает в своем блоге по адресу *http://regex.info/blog/2006-09-15/247*.

> Когда кто-то, сталкиваясь с проблемой, думает: «А-а, понимаю, здесь поможет регулярное выражение», то в результате он получает две проблемы.

Вторая проблема, которую получают те, кто так думает, состоит в том, что они не читали справочное руководство, которое вы держите в руках. Продолжайте читать его. Регулярные выражения – это очень мощный инструмент. Если решаемая задача связана с обработкой или извлечением текста на компьютере, то владение регулярными выражениями поможет вам сэкономить массу времени.

Множество диалектов регулярных выражений

Итак, заголовком предыдущего раздела мы ввели вас в заблуждение. Мы не дали определение регулярных выражений. И мы не можем сделать этого. В мире не существует официального стандарта, который точно определял бы, какой текстовый шаблон является регулярным выражением, а какой нет. Как вы можете себе представить, каждый, кто проектирует свой язык программирования, и каждый разработчик приложений, обрабатывающих тексты, имеет свое представление о том, какими должны быть регулярные выражения. Поэтому в результате мы имеем массу *диалектов* регулярных выражений.

К счастью, большинство проектировщиков и разработчиков оказываются достаточно ленивы. Зачем создавать нечто совершенно новое, когда можно скопировать то, что уже существует? В результате почти все современные диалекты регулярных выражений, включая и те, что рассматриваются в этой книге, ведут свою историю от языка программирования Perl. Мы называем эти диалекты *регулярными выражениями в стиле языка Perl*. Синтаксис этих диалектов прост и в основном совместим, хотя и не полностью.

Писатели тоже ленивы. Единственное регулярное выражение мы обычно называем *regex* или *regexp*, а термин *regexes* используем для обозначения множественного числа.

Диалекты регулярных выражений не связаны жестко с какими-то определенными языками программирования. Языки сценариев обычно имеют свои собственные встроенные диалекты регулярных выражений. Другие языки программирования опираются на использование библиотек, реализующих поддержку регулярных выражений. Некоторые библиотеки доступны для нескольких языков программирования, а некоторые языки программирования предоставляют возможность выбора из нескольких библиотек.

Эта вводная глава рассматривает диалекты регулярных выражений без всякой связи с языками программирования. Листинги с исходными текстами начнут появляться в главе 3; в разделе «Языки программирования и диалекты регулярных выражений» вы узнаете, с какими диалектами вам предстоит работать. А пока оставим в стороне все вопросы, связанные с программированием. Инструменты, перечисленные в сле-

дующем разделе, обеспечивают простоту исследования синтаксиса регулярных выражений, «обучая в процессе работы».

Диалекты регулярных выражений, рассматриваемые в этой книге

Для этой книги мы отобрали наиболее распространенные диалекты регулярных выражений, используемых в наши дни. Все они являются диалектами регулярных выражений языка Perl. Некоторые диалекты имеют более широкие возможности по сравнению с другими. Но, как правило, когда два диалекта обладают одинаковым набором возможностей, они используют один и тот же синтаксис. Мы укажем на некоторые из досадных несовместимостей, с которыми нам пришлось столкнуться.

Все рассматриваемые здесь диалекты являются частью какого-либо языка программирования или библиотеки, которые продолжают активно разрабатываться. Из списка диалектов, что приводится ниже, вы узнаете, какие версии рассматриваются в этой книге. На протяжении всей книги мы будем упоминать диалекты без их версий, если представленные регулярные выражения одинаково работают во всех диалектах, что случается практически всегда. Помимо исправлений ошибок, проявляющихся в крайних случаях, диалекты обычно не изменяются, за исключением добавления новых особенностей и введения новых синтаксических конструкций, которые в предыдущих версиях трактовались как ошибка.

.NET

Платформа Microsoft .NET Framework обеспечивает поддержку полноценного диалекта регулярных выражений в стиле языка Perl в виде пакета `System.Text.RegularExpressions`. В этом издании описываются версии .NET начиная с 1.0 по 4.0. Строго говоря, существует всего две версии диалекта .NET: 1.0 и 2.0. В версиях .NET 1.1, 3.0 и 3.5 классы Regex не претерпели никаких изменений. В версии 4.0 класс Regex получил несколько новых методов, но синтаксис регулярных выражений не изменился.

Любые языки программирования, поддерживаемые платформой .NET, включая C#, VB.NET, Delphi для .NET и даже COBOL.NET, обладают полным доступом ко всем особенностям диалекта регулярных выражений .NET. Если приложение, разработанное на основе платформы .NET, обеспечивает поддержку регулярных выражений, можно быть совершенно уверенным, что оно использует диалект .NET, даже если заявлено, что используются «регулярные выражения Perl». Долгое время исключение составляла сама среда разработки Visual Studio. Вплоть до версии Visual Studio 2010 в интегрированной среде разработки (IDE) использовался устаревший диалект регулярных выражений, реализованный с самого начала, который вообще не является диалектом в стиле языка Perl. И только в версии Visual Studio 11 (которая была в состоянии бета-версии на

момент написания этих строк) наконец-то стал использоваться диалект регулярных выражений .NET и в самой IDE.

Java

Java 4 – это первая версия, в которой присутствует встроенная поддержка регулярных выражений, реализованная в виде пакета `java.util.regex`. Эта реализация быстро затмила различные сторонние библиотеки для Java. Кроме того, что этот пакет является стандартным и встроенным, он также предлагает полноценную реализацию Perl-совместимого диалекта регулярных выражений и имеет превосходную производительность, сопоставимую даже с приложениями, написанными на языке C. В этой книге рассматривается пакет `java.util.regex`, входящий в состав Java 4, 5, 6 и 7.

Если вы пользуетесь программным обеспечением, написанным на языке Java не более чем пару лет тому назад, любые регулярные выражения, поддерживаемые им, весьма вероятно используют диалект Java.

JavaScript

Название *JavaScript* используется в этой книге для обозначения диалекта регулярных выражений, определяемого версиями 3 или 5 стандарта ECMA-262. Этот стандарт определяет язык программирования ECMAScript, который в различных веб-браузерах более широко известен под названиями JavaScript и JScript. Все браузеры – Internet Explorer (начиная с версии 5.5), Firefox, Chrome, Opera и Safari – реализуют третью или пятую версию стандарта ECMA-262. Однако у каждого из браузеров имеются свои отклонения от стандарта. Мы будем указывать на эти отклонения в ситуациях, когда это будет иметь значение.

Если веб-сайт позволяет выполнять поиск или фильтрацию содержимого с помощью регулярных выражений без необходимости ожидать ответ веб-сервера, следовательно, он использует диалект JavaScript регулярных выражений, который поддерживается браузерами на стороне клиента. Этот диалект используется даже в языках программирования Microsoft VBScript и Adobe ActionScript 3, хотя в ActionScript 3 в него добавлены некоторые дополнительные особенности.

XRegExp

XRegExp – это свободно распространяемая библиотека JavaScript, разработанная Стивеном Левитаном (Steven Levithan). Ее можно получить по адресу *http://xregexp.com*. Библиотека XRegExp расширяет синтаксис регулярных выражений JavaScript и устраняет некоторые несовместимости между браузерами. Рецепты в этой книге, использующие особенности регулярных выражений, не доступные в стандартном диалекте JavaScript, содержат дополнительные решения, реализованные с использованием XRegExp. Если в решении указывается диалект XRegExp, это означает, что оно будет работать

в стандартном JavaScript при использовании библиотеки XRegExp и не будет – в JavaScript без библиотеки XRegExp. Если в решении указывается диалект JavaScript, следовательно, оно будет работать в сценариях на JavaScript независимо от того, используется библиотека XRegExp или нет.

Эта книга охватывает версию XRegExp 2.0. Решения в рецептах предполагают использование сценария `xregexp-all.js` и соответственно доступность полноценной поддержки Юникода в библиотеке XRegExp.

PCRE

PCRE – это библиотека реализации «Perl-Compatible Regular Expressions» (Perl-совместимых регулярных выражений), написанная Филипом Хейзелем (Philip Hazel) на языке C. Получить исходные тексты библиотеки можно по адресу *http://www.pcre.org*. В этой книге рассматриваются версии PCRE с 4 по 8.

Несмотря на утверждение, что библиотека PCRE совместима с диалектом регулярных выражений языка Perl, что в отношении ее даже более справедливо, чем для других диалектов, тем не менее она всего лишь поддерживает стиль языка Perl. Некоторые особенности, такие как поддержка Юникода, в ней отличаются, и она не допускает внедрение программного кода на языке Perl в регулярные выражения, как это допускается в самом языке Perl.

Благодаря свободной лицензии и надежности библиотека PCRE нашла применение во многих языках программирования и приложениях. Она встроена в язык PHP и включена в состав многих компонентов Delphi. Если заявлено, что приложение поддерживает «Perl-совместимые» регулярные выражения без указания конкретного используемого диалекта, наиболее вероятно, что используется библиотека PCRE.

Perl

То, что язык Perl имеет встроенную поддержку регулярных выражений, является главной причиной их популярности в наши дни. В этой книге рассматриваются версии Perl 5.6, 5.8, 5.10, 5.12 и 5.14. Каждая из этих версий добавляет новые особенности в синтаксис регулярных выражений Perl. Когда в книге указывается, что регулярное выражение работает в определенной версии Perl, это означает, что оно будет работать также во всех последующих версиях, охватываемых этой книгой.

Многие приложения и библиотеки реализации регулярных выражений, которые, как утверждается, поддерживают регулярные выражения языка Perl или совместимы с ними, на самом деле просто используют регулярные выражения в стиле языка Perl. Они поддерживают синтаксис, похожий на синтаксис Perl, но не поддерживают тот же набор особенностей регулярных выражений. Наиболее вероятно,

что они используют один из диалектов, что приводятся в списке ниже. Все эти диалекты являются диалектами в стиле языка Perl.

Python

Язык программирования Python обеспечивает поддержку регулярных выражений посредством модуля `re`. В этой книге рассматриваются версии Python с 2.4 по 3.2. Различия в поддержке регулярных выражений между версиями Python 2.4, 2.5, 2.6 и 2.7 весьма незначительны. В версии Python 3.0 была существенно улучшена поддержка Юникода в регулярных выражениях, а версии 3.1 и 3.2 не привнесли в регулярные выражения никаких изменений.

Ruby

Поддержка регулярных выражений в языке Ruby реализована как часть самого языка, так же как и в Perl. В этой книге рассматриваются версии Ruby 1.8 и 1.9. Версия Ruby 1.8 по умолчанию использует диалект регулярных выражений, реализация которого включена непосредственно в исходные тексты Ruby. Версия Ruby 1.9 по умолчанию использует библиотеку регулярных выражений Onigurama. Имеется возможность скомпилировать Ruby 1.8 с поддержкой библиотеки Onigurama, а Ruby 1.9 – с поддержкой старого диалекта регулярных выражений Ruby. В этой книге мы будем обозначать встроенный диалект Ruby как Ruby 1.8, а диалект Onigurama – как Ruby 1.9.

Чтобы определить, какой диалект регулярных выражений Ruby используется у вас, попробуйте выполнить регулярное выражение `‹a++›`. Если поддерживается диалект Ruby 1.8, будет выдано сообщение об ошибке, потому что он не поддерживает захватывающие квантификаторы, тогда как в диалекте Ruby 1.9 этому выражению соответствует строка, состоящая из одного или более символов.

Библиотека Onigurama предусматривает обратную совместимость с диалектом Ruby 1.8, просто добавляя новые возможности, которые не влияют на работу существующих регулярных выражений. Разработчики даже оставили некоторые особенности, которые следовало бы изменить, например выражение `‹(?m)›` означает «точка соответствует концу строки», тогда как в других диалектах для этой же цели используется выражение `‹(?s)›`.

Поиск с заменой с помощью регулярных выражений

Поиск с заменой – типичная задача, которая решается с применением регулярных выражений. Функция, реализующая поиск с заменой, принимает строку, в которой производится поиск, регулярное выражение и строку замены, которая будет замещать найденные фрагменты. Результатом функции является строка, где все участки, соответствующие регулярному выражению, заменяются замещающим текстом.

Хотя сам замещающий текст не является регулярным выражением, допускается использовать специальный синтаксис с целью динамического создания замещающего текста. Как это делается, описывается в рецептах 2.20 и 2.21. Некоторые диалекты также поддерживают вставку совпавшего содержимого в замещающий текст, как показано в рецепте 2.22. В главе 3 в рецепте 3.16 показано, как с помощью программного кода генерировать различные замещающие тексты для каждого совпадения.

Множество диалектов определения замещающего текста

Многообразие идей, выдвигаемых различными разработчиками реализаций регулярных выражений, привело к появлению широкого диапазона диалектов регулярных выражений, каждый из которых обладает своими особенностями. История выработки синтаксиса определения замещающего текста также не является исключением. В действительности, диалектов определения замещающего текста оказалось даже больше, чем диалектов самих регулярных выражений. Создание механизма регулярных выражений – задача не из легких. Большинство программистов предпочитает использовать один из существующих механизмов регулярных выражений и затем прикручивать к нему функцию поиска с заменой, что совсем несложно. В результате появилось множество диалектов определения замещающего текста для библиотек поддержки регулярных выражений, в которых отсутствуют собственные реализации функции поиска с заменой.

К счастью, все диалекты регулярных выражений, представленные в этой книге, обладают собственными возможностями выполнения поиска с заменой, за исключением PCRE. Этот недостаток PCRE усложняет жизнь тем программистам, которые используют диалекты, созданные на его основе. Свободно распространяемая библиотека PCRE не содержит встроенных функций, реализующих возможность замены. Поэтому все приложения и языки программирования, опирающиеся на использование PCRE, должны предоставлять собственные реализации функции поиска с заменой. Многие программисты пытаются копировать существующие реализации, но практически всегда делают это несколько по-разному.

Ниже перечислены диалекты определения замещающего текста, рассматриваемые в этой книге. За дополнительной информацией о соответствующих им диалектах регулярных выражений обращайтесь к разделу «Множество диалектов регулярных выражений».

.NET

Пакет `System.Text.RegularExpressions` содержит несколько функций, реализующих поиск с заменой. Диалект определения замещающего текста в .NET соответствует диалекту регулярных выражений .NET. Все версии .NET используют один и тот же диалект определения

замещающего текста. Новые особенности регулярных выражений, появившиеся в .NET 2.0, никак не повлияли на синтаксис определения замещающего текста.

Java

Пакет `java.util.regex` содержит собственные встроенные функции поиска с заменой. В этой книге рассматриваются версии Java 4, 5, 6 и 7.

JavaScript

Под названием *JavaScript* в этой книге мы будем подразумевать как диалект определения замещающего текста, так и диалект регулярных выражений, определяемый версиями 3 и 5 стандарта ECMA-262.

XRegExp

Библиотека XRegExp Стивена Левитана имеет собственную функцию `replace()`, устраняющую несовместимость браузеров и добавляющую поддержку *обратных ссылок* (*backreferences*) на именованные группы, поддерживаемые в диалекте XRegExp. Рецепты в этой книге, использующие именованные группы, включают дополнительные решения на диалекте XRegExp. Если в решении указывается, что используется диалект XRegExp определения текста замены, это означает, что оно будет работать в JavaScript только при использовании библиотеки XRegExp. Если в решении указывается диалект JavaScript, следовательно, оно будет работать в сценариях на JavaScript независимо от того, используется ли библиотека XRegExp или нет.

Эта книга охватывает версию XRegExp 2.0, которую можно загрузить по адресу *http://xregexp.com*.

PHP

В этой книге рассматривается реализация поиска с заменой на языке PHP в виде функции `preg_replace`. Данная функция использует диалект регулярных выражений PCRE и диалект PHP определения замещающего текста. Впервые она появилась в версии PHP 4.0.0.

В других языках программирования, опирающихся на библиотеку PCRE, используются отличные от PHP диалекты определения замещающего текста. В зависимости от того, где черпали вдохновение разработчики того или иного языка программирования, синтаксис определения замещающего текста может быть похожим на диалект PHP или на любой другой диалект определения замещающего текста из представленных в этой книге.

В языке PHP имеется также функция `ereg_replace`. Эта функция использует другой диалект регулярных выражений (POSIX ERE) и иной диалект определения замещающего текста. Семейство функций `ereg` из языка PHP не будет рассматриваться в этой книге.

Perl

Язык Perl обладает встроенной поддержкой замещения текста с помощью регулярных выражений, реализованной в виде инструкции

s/regex/replace/. Диалект определения замещающего текста в языке Perl соответствует его диалекту регулярных выражений. В этой книге рассматриваются версии от Perl 5.6 до Perl 5.14. В Perl 5.10 была добавлена поддержка именованных обратных ссылок в замещающем тексте как дополнение к добавленной в регулярные выражения поддержке именованных групп.

Python

Модуль re в языке Python реализует поиск с заменой в виде функции sub. Диалект определения замещающего текста в языке Python соответствует диалекту регулярных выражений Python. В этой книге рассматриваются версии Python с 2.4 по 3.2. Различия в синтаксисе определения текста замены между этими версиями отсутствуют.

Ruby

Поддержка регулярных выражений в языке Ruby встроена непосредственно в сам язык, включая и функцию поиска с заменой. В этой книге рассматриваются версии Ruby 1.8 и 1.9. Если синтаксис регулярных выражений в версиях Ruby 1.8 и 1.9 имеет существенные отличия, то синтаксис определения замещающего текста остается практически одинаковым, за исключением того, что в версии Ruby 1.9 добавлена поддержка именованных обратных ссылок в замещающем тексте. Именованные группы – это новая особенность, появившаяся в поддержке регулярных выражений Ruby 1.9.

Инструменты для работы с регулярными выражениями

Если у вас нет опыта программирования регулярных выражений, мы рекомендуем для начала поэкспериментировать с ними с помощью дополнительных инструментов и только потом включать их в программный код. В этой главе представлены примеры простых регулярных выражений, которые не содержат дополнительных экранированных последовательностей, необходимых при использовании в языках программирования (даже в языке командной оболочки UNIX). Эти регулярные выражения можно вводить непосредственно в поле поиска приложения.

В главе 3 описывается, как оформляются регулярные выражения в исходных текстах программ. Оформление регулярных выражений в виде строк еще больше осложняет их восприятие человеком, потому что к правилам экранирования последовательностей внутри регулярных выражений добавляются правила экранирования служебных символов в строках. Мы не будем касаться этой проблемы до рецепта 3.1. Как только вы разберетесь с основами регулярных выражений, вы сможете увидеть лес за частоколом обратных слэшей.

Инструменты, описываемые в этом разделе, кроме всего прочего обеспечивают возможность отладки, проверки синтаксиса и другие удобства,

не доступные в большинстве сред программирования. По этой причине в процессе разработки регулярных выражений для своих приложений вы можете найти для себя удобным сначала создать сложное регулярное выражение в одном из этих инструментов и потом перенести его в свою программу.

RegexBuddy

Программа RegexBuddy (рис. 1.1) – один из наиболее полнофункциональных инструментов, доступных на момент написания этих строк, позволяющий создавать, тестировать и реализовывать регулярные выражения. Он обладает уникальной возможностью имитировать все диалекты регулярных выражений, рассматриваемые в этой книге, и даже преобразовывать их из одного диалекта в другой.

Программа RegexBuddy была спроектирована и разработана Яном Гойвертсом (Jan Goyvaerts), одним из авторов этой книги. Работа над программой RegexBuddy сделала Яна экспертом в области регулярных выражений, а использование RegexBuddy помогло довести его соавтора Стивена до этапа передачи этой книги в издательство O'Reilly.

Если интерфейс программы (рис. 1.1) покажется вам перегруженным, то это только потому, что мы распахнули большую часть из имеющихся панелей, чтобы продемонстрировать обширные возможности RegexBuddy. По умолчанию все панели аккуратно свернуты в строку вкладок. Существует также возможность перемещать панели на второй монитор.

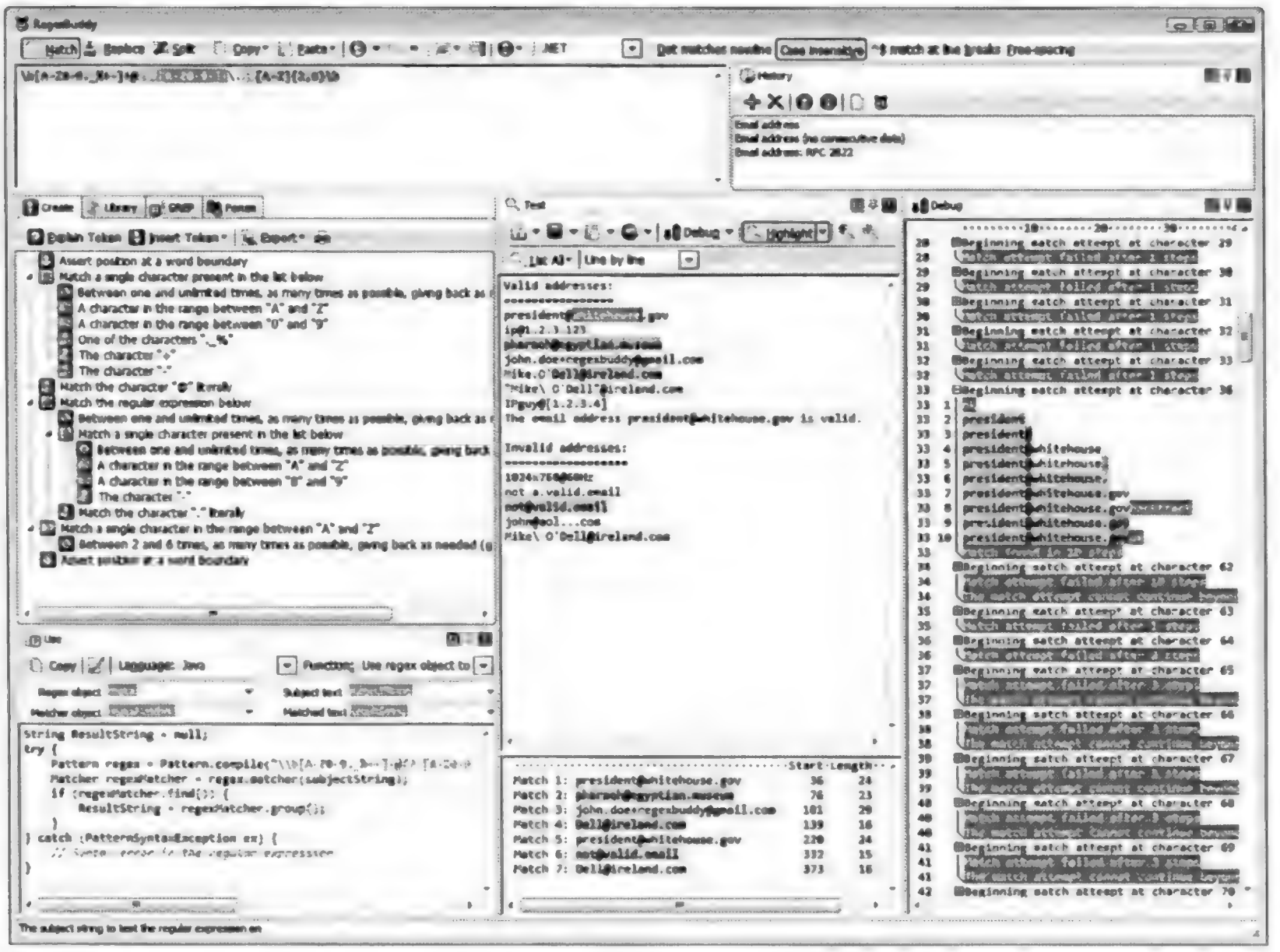


Рис. 1.1. RegexBuddy

Чтобы опробовать какое-либо регулярное выражение из представленных в этой книге, просто введите его в поле редактирования, расположенное в верхней части окна RegexBuddy. Программа автоматически подсветит синтаксис введенного регулярного выражения, ярко выделив возможные ошибки и непарные скобки.

По мере ввода регулярного выражения в панели Create (Создать) автоматически будет создаваться подробный отчет на английском языке. Двойной щелчок мышью на любом из описаний в дереве будет вызывать переход к редактированию соответствующей части регулярного выражения. Вы можете вставлять новые элементы в регулярное выражение вручную или, щелкнув на кнопке Insert Token (Вставить лексему), выбирать нужный элемент из меню. Например, если вы не помните достаточно сложный синтаксис положительных опережающих проверок, можно попросить RegexBuddy вставить соответствующие символы.

После ввода или вставки некоторого образца текста в панель Test (Проверка) при нажатой кнопке Highlight (Выделить) программа RegexBuddy автоматически выделит фрагменты текста, соответствующие регулярному выражению.

В число кнопок, которые вам наверняка придется использовать чаще всего, входят:

List All (Перечислить все)

Отображает список всех совпадений.

Replace (Заменить)

Кнопка Replace (Заменить), расположенная в верхней части окна, дает возможность ввести замещающий текст. Кнопка Replace (Заменить) в панели Test (Проверка) дает возможность увидеть, как будет выглядеть текст после выполнения операции замены.

Split (Разделить) (кнопка в панели Test, а не в верхней части окна)

Интерпретирует регулярное выражение как разделитель и разбивает исходный текст на лексемы по совпадениям, найденным с помощью регулярного выражения.

Щелкните на любой из этих кнопок и в меню List All (Перечислить все) выберите пункт Update Automatically (Обновлять автоматически), чтобы обеспечить автоматическую синхронизацию результатов по мере редактирования регулярного выражения или текста примера.

Чтобы в точности увидеть, как работает (или не работает) регулярное выражение, следует щелкнуть в панели Test (Проверка) на выделенном совпадении или в месте, где обнаружено отсутствие совпадения, и затем щелкнуть на кнопке Debug (Отладка). Программа RegexBuddy переключится в панель Debug (Отладка) и по шагам продемонстрирует процесс поиска совпадения. Щелкнув мышью в пределах вывода отладчика, можно определить, какому фрагменту регулярного выражения соответствует текст, на котором был выполнен щелчок. Щелкнув мышью в пределах

регулярного выражения, можно выделить фрагмент текста в панели отладчика, соответствующий данной части регулярного выражения.

В панели Use (Использование) можно выбрать предпочтительный язык программирования и функцию и тут же получить исходный программный код реализации регулярного выражения. Исходные тексты шаблонов для RegexBuddy полностью доступны для редактирования во встроенном редакторе шаблонов. Благодаря этому можно добавлять новые функции и даже новые языки программирования или изменять существующие.

Чтобы испытать работу регулярного выражения на большом объеме данных, можно переключиться в панель GREP и произвести поиск (возможно, с заменой) в любом числе файлов и каталогов.

Обнаружив регулярное выражение в исходном тексте сопровождаемой программы, можно скопировать его в буфер обмена, включая ограничивающие его кавычки или слэши, затем в окне программы RegexBuddy щелкнуть на кнопке Paste (Вставить), расположенной в верхней части, и выбрать стиль оформления строк для вашего языка программирования. После этого регулярное выражение появится в окне RegexBuddy как обычное регулярное выражение, без дополнительных кавычек и экранирующих последовательностей, необходимых для оформления литералов строк. Чтобы создать в буфере обмена строку с соблюдением синтаксических правил, достаточно щелкнуть на кнопке Copy (Копировать), расположенной в верхней части, которую затем можно будет вставить в исходный текст программы.

По мере накопления опыта в панели Library (Библиотека) можно создавать собственную библиотеку регулярных выражений. Не забывайте добавлять подробное описание и тестовые примеры при сохранении регулярного выражения. Регулярные выражения могут выглядеть сложными для понимания даже для опытных экспертов.

В случае неудачи при создании собственного регулярного выражения можно щелкнуть на вкладке панели Forum (Форум) и затем на кнопке Login (Зарегистрироваться). Если вы приобрели программу RegexBuddy на законных основаниях, на экране появится форма входа, заполнив которую и щелкнув на кнопке OK, вы подключитесь к форуму пользователей RegexBuddy. Стивен и Ян часто бывают здесь.

Программа RegexBuddy работает под управлением операционных систем Windows 98, ME, 2000, XP, Vista, 7 и 8. Пользователи Linux и Apple могут запускать ее под управлением VMware, Parallels, CrossOver Office и, с некоторыми усилиями, под WINE. Загрузить оценочную версию RegexBuddy можно по адресу *http://www.regexbuddy.com/RegexBuddyCookbook.exe*. Оценочная версия в течение семи дней фактического использования обладает полным набором функциональных возможностей, за исключением доступа к форуму.

RegexPal

RegexPal (рис. 1.2) – это веб-приложение, выполняющее тестирование регулярных выражений. Оно было написано Стивеном Левитаном (Steven Levithan), одним из авторов этой книги. Все, что необходимо для работы с этим приложением, – это современный веб-браузер. Приложение RegexPal полностью написано на языке JavaScript. Вследствие этого оно поддерживает только диалект регулярных выражений JavaScript, реализованный в веб-браузере, используемом для доступа к веб-приложению.

Рис. 1.2. RegexPal

Чтобы опробовать какое-либо регулярное выражение из этой книги, откройте страницу *http://www.regexpal.com* и введите регулярное выражение в поле вверху. Приложение RegexPal автоматически подсветит синтаксис введенного регулярного выражения, что сразу же позволит определить синтаксические ошибки. Приложение RegexPal учитывает проблемы несовместимости браузеров, которые могут нарушить ваши планы при разработке регулярных выражений для JavaScript. Если какая-то синтаксическая конструкция работает некорректно в некоторых браузерах, RegexPal выделит ее как ошибку.

Теперь можно ввести или вставить текст примера в большое поле в центре, после чего RegexPal автоматически выделит фрагменты текста, соответствующие регулярному выражению.

Здесь нет никаких кнопок, на которых требовалось бы щелкать, что делает RegexPal одним из самых удобных веб-приложений тестирования регулярных выражений.

RegexMagic

RegexMagic (рис. 1.3) – еще один инструмент, спроектированный и разработанный Яном Гойвертсом. Если приложение RegexBuddy упрощает работу с синтаксисом регулярных выражений, то RegexMagic в первую очередь предназначено для тех, кто не желает разбираться в этом синтаксисе и уж точно не хочет читать 500-страничные книги по этой теме.

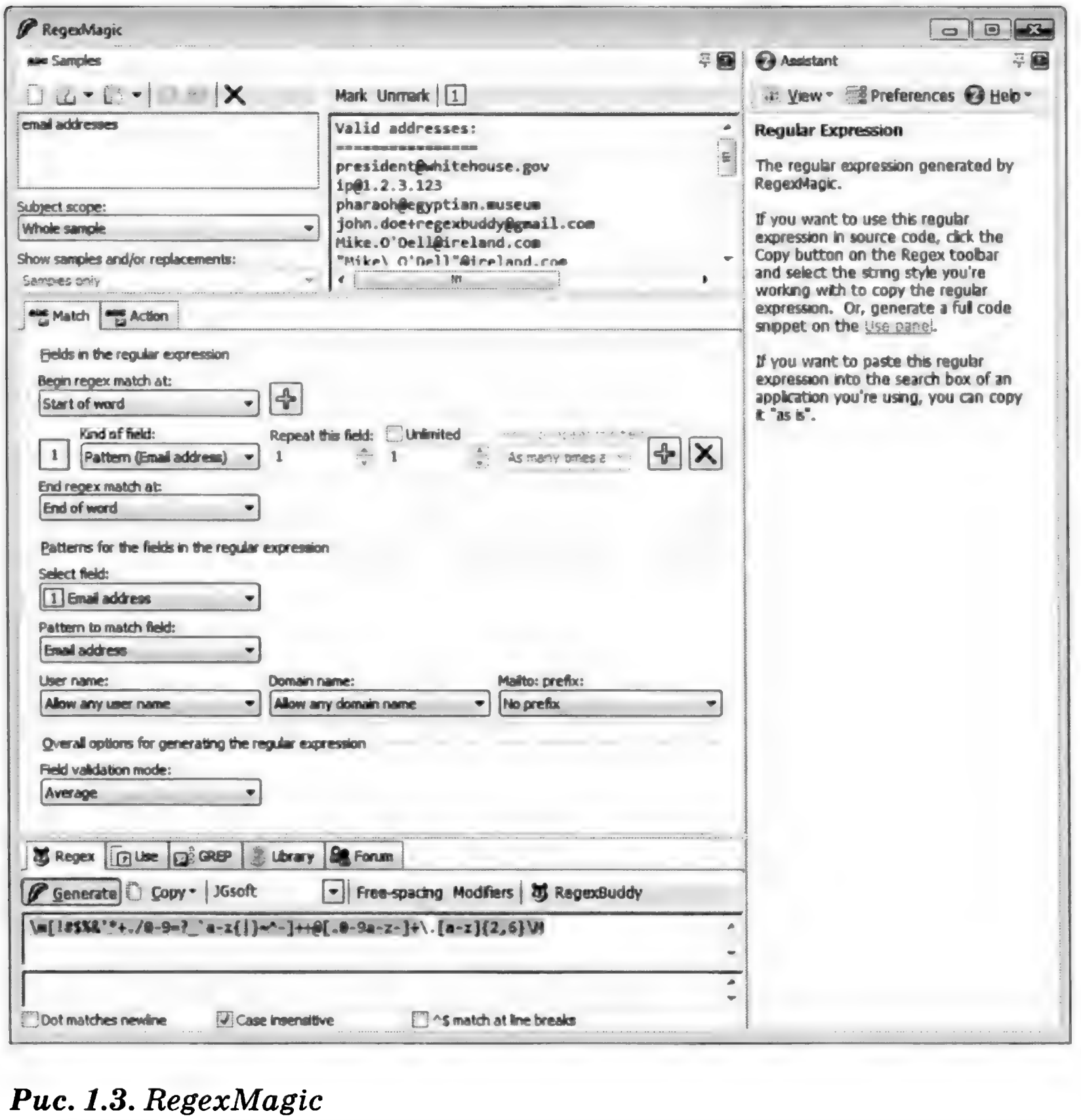


Рис. 1.3. RegexMagic

При использовании RegexMagic вы описываете, какой результат требуется получить, опираясь на образец исходного текста и высокоуровневые шаблоны RegexMagic. На рис. 1.3 показано, что для получения регулярного выражения, соответствующего адресам электронной почты, достаточно просто выбрать шаблон «email address» (адрес электронной почты). Имеется возможность выполнить дополнительные настройки

шаблона, ограничив совпадения только с определенными именами пользователей и доменов, а также можно указать допустимость или обязательность префикса `mailto:`.

Раз вы читаете эту книгу, значит, вы уже выбрали для себя путь к полному овладению регулярными выражениями. Поэтому RegexMagic едва ли станет вашим основным инструментом. Но иногда могут возникать ситуации, когда он может оказаться полезным. В рецепте 6.7 мы расскажем, как создавать регулярные выражения, соответствующие диапазонам чисел. Регулярные выражения – не лучший способ выяснить, попадает ли некоторое число в заданный диапазон, но иногда они могут оказаться единственным доступным инструментом. Приложений со встроенной поддержкой регулярных выражений намного больше, чем приложений со встроенной поддержкой языков сценариев. Прием, описываемый в рецепте 6.7, несложен. Но реализация его вручную может оказаться весьма утомительным занятием.

Представьте, что вместо простых примеров, которые приводятся в рецепте 6.7, вам потребовалось найти числа в диапазоне от 2 147 483 648 (2^{31}) до 4 294 967 295 ($2^{32}-1$) в десятичной нотации. Пользуясь приложением RegexMagic, достаточно просто выбрать шаблон «Integer» (целое число), отметить вариант «decimal» (десятичное) и указать границы диапазона `2147483648..4294967295`. В «строгом» режиме RegexMagic немедленно сгенерирует следующее страшилище:

```
\b(?:429496729[0-5]|42949672[0-8][0-9]|4294967[01][0-9]{2}|429496[0-6]↵
[0-9]{3}|42949[0-5][0-9]{4}|4294[0-8][0-9]{5}|429[0-3][0-9]{6}|42[0-8]↵
[0-9]{7}|4[01][0-9]{8}|3[0-9]{9}|2[2-9][0-9]{8}|21[5-9][0-9]{7}|214[89]↵
[0-9]{6}|2147[5-9][0-9]{5}|214749[0-9]{4}|214748[4-9][0-9]{3}|2147483↵
[7-9][0-9]{2}|21474836[5-9][0-9]|214748364[89])\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Программа RegexMagic работает под управлением операционных систем Windows 98, ME, 2000, XP, Vista, 7 и 8. Пользователи Linux и Apple могут запускать ее под управлением VMware, Parallels, CrossOver Office и, с некоторыми усилиями, под WINE. Загрузить оценочную версию RegexMagic можно по адресу *http://www.regexmagic.com/RegexMagicCookbook.exe*. Оценочная версия в течение семи дней фактического использования обладает полным набором функциональных возможностей, за исключением доступа к форуму.

Другие примеры веб-приложений тестирования регулярных выражений

Достаточно легко создать простое веб-приложение для тестирования регулярных выражений. Если у вас имеются основные навыки веб-разработки, в главе 3 вы найдете все необходимое для создания такого приложения. Сотни людей уже сделали это, причем некоторые из них до-

полнили свои приложения возможностями, которые делают их заслуживающими внимания.

RegexPlanet

RegexPlanet – это сайт, разработанный Эндрю Маркусом (Andrew Marcuse). Он утверждает, что его приложение позволяет тестировать регулярные выражения с гораздо большим количеством разнообразных библиотек поддержки регулярных выражений, чем любое другое. На домашней странице веб-приложения вы найдете ссылки на страницы тестирования для диалектов Java, JavaScript, .NET, Perl, PHP, Python и Ruby. Все они предоставляют простой и одинаковый интерфейс, отличаясь лишь списками параметров, доступных в разных языках программирования. На рис. 1.4 изображена версия для диалекта .NET.

Введите или скопируйте регулярное выражение в поле Regular expression (Регулярное выражение). Если вы хотите опробовать поиск с заменой, вставьте текст замены в поле Replacement (Замена). Протестировать введенное регулярное выражение можно с любым количеством испытуемых строк. Скопируйте испытуемые строки в поля Input (Исходная строка). Щелкните на кнопке More inputs (Добавить исходные строки), если потребуется испытать более пяти исходных строк. В поля Regular expression (Регулярное выражение) и Input (Исходная строка) допускается вводить или вставлять многострочный текст, несмотря на то, что на экране видна только одна строка. Стрелки «вверх» и «вниз» справа от полей ввода позволяют прокручивать многострочный текст.

Закончив ввод, щелкните на кнопке Test (Проверить), чтобы отправить все строки на сервер *regexplanet.com*. В результате вы получите страницу, как показано на рис. 1.4, со списком результатов тестирования вверху. Первые две колонки повторяют ваш ввод, остальные содержат результаты вызовов различных функций. Эти колонки отличаются для разных языков программирования, поддерживаемых сайтом.

regex.larsolavtorvik.com

Ларс Олав Торвик (Lars Olav Torvik) создал небольшое, но замечательное приложение тестирования регулярных выражений, доступное по адресу *http://regex.larsolavtorvik.com* (рис. 1.5).

Для начала выберите требуемый диалект регулярных выражений, щелкнув на названии диалекта в верхней части страницы. Ларс предлагает выбор между диалектами PHP PCRE, PHP POSIX и JavaScript. PHP PCRE – это диалект PCRE, рассматриваемый в этой книге и используемый в языке PHP семейством функций preg. POSIX – это устаревший диалект с ограниченными возможностями, используемый семейством функций ereg в языке PHP, который не рассматривается в данной книге. В случае выбора JavaScript вы будете работать с реализацией JavaScript в вашем браузере.

Рис. 1.4. *RegexPlanet*

Введите регулярное выражение в поле с меткой Pattern (Шаблон) и текст примера в поле Subject (Испытуемый текст). Спустя мгновение в поле Matches (Совпадения) появится текст примера с выделенными фрагментами, соответствующими регулярному выражению. В поле Code (Программный код) появится строка программного кода, которая применяет регулярное выражение к испытуемому тексту. Операция копирования

и вставки этой строки в текстовом редакторе позволит вам сэкономить время на преобразование вручную регулярного выражения в строковый литерал. Любая строка или массив, возвращаемые программным кодом, отображаются в поле Result (Результат). Так как при создании своего веб-приложения Ларс использовал технологию Ajax, результаты появляются всего через несколько мгновений для любого выбранного диалекта. Для работы с этим инструментом необходимо подключение к сети, так как интерпретатор PHP выполняется на сервере, а не в браузере.

Рис. 1.5. *regex.larsolavtorvik.com*

Во второй колонке отображается список команд и параметров регулярных выражений. Перечень доступных команд и параметров зависит от выбранного диалекта регулярных выражений. В число команд обычно входят такие операции, как поиск совпадений, замена и разбиение. В число параметров входят такие, как чувствительность к регистру символов, а также ряд других параметров, характерных для реализации. Эти команды и параметры описываются в главе 3.

Nregex

http://www.nregex.com (рис. 1.6) – это простое веб-приложение, созданное Дэвидом Сераянгом (David Seruyange), предназначено для тестирования регулярных выражений. Оно поддерживает диалект .NET 2.0, который также используется в версиях .NET 3.0, 3.5 и 4.0.

Рис. 1.6. *Nregex*

Расположение элементов на странице может немного сбивать с толку. Регулярное выражение следует вводить в поле с надписью Regular Expression (Регулярное выражение), а параметры выражения определяются с помощью флажков под ним. Испытуемый текст вводится в поле, расположенное ниже, где первоначально находится текст «`If I just had $5.00 then "she" wouldn't be so @#$! mad.`». Если предметом испытаний является веб-страница, можно ввести адрес URL в поле Load Target From URL (Загру-

зить испытуемый текст с адреса URL) и щелкнуть на кнопке Load (Загрузить), расположенной под полем ввода. Если предметом испытаний является файл, расположенный на жестком диске, можно щелкнуть на кнопке Browse (Просмотреть), отыскать требуемый файл и щелкнуть на кнопке Load (Загрузить), расположенной под полем ввода.

Испытуемый текст появится в поле Matches & Replacements (Совпадения и замены), расположенном в центре страницы, в котором будут выделены фрагменты, соответствующие регулярному выражению. Если ввести какой-либо текст в поле Replacement String (Строка замены), будет отображен результат выполнения операции поиска с заменой. В случае наличия ошибки в регулярном выражении будет выведено многоточие (...).

Поиск соответствий регулярному выражению производится программным кодом платформы .NET, выполняющимся на стороне сервера, поэтому для работы с сайтом необходимо подключение к сети. Если автоматическое обновление страницы выполняется слишком медленно, наиболее вероятно, что это происходит из-за большого объема испытуемого текста. В этом случае отметьте флажок Manually Evaluate Regex (Производить вычисления вручную), расположенный выше поля ввода регулярного выражения. В результате на странице появится кнопка Evaluate (Вычислить), щелчок на которой будет приводить к обновлению содержимого поля Matches & Replacements (Совпадения и замены).

Rubular

Майкл Ловитт (Michael Lovitt) разместил минималистичное приложение для испытания регулярных выражений по адресу *http://www.rubular.com* (рис. 1.7). На момент написания этих строк оно позволяло выбирать между Ruby 1.8.7 и Ruby 1.9.2, что дает возможность тестировать регулярные выражения в диалектах Ruby 1.8 и Ruby 1.9, описываемых в этой книге.

Регулярное выражение вводится в поле Your regular expression (Ваше регулярное выражение), расположенное между двумя символами слэша. Можно отключить чувствительность к регистру символов, введя символ `i` в небольшое поле ввода, находящееся правее второго символа слэша. При желании точно так же можно определить параметр «точка соответствует концу строки», введя символ `m` в это поле. Комбинация символов `im` активизирует оба параметра. Эти соглашения могут показаться недружественными для тех, кто не знаком с языком Ruby, однако они соответствуют синтаксису */regex/im*, используемому в исходных текстах программ на языке Ruby.

Введите или скопируйте испытуемый текст в поле Your test string (Ваша испытуемая строка) и подождите мгновение. В поле Match result (Результат поиска соответствия) появится испытуемый текст с выделенными фрагментами, соответствующими регулярному выражению.

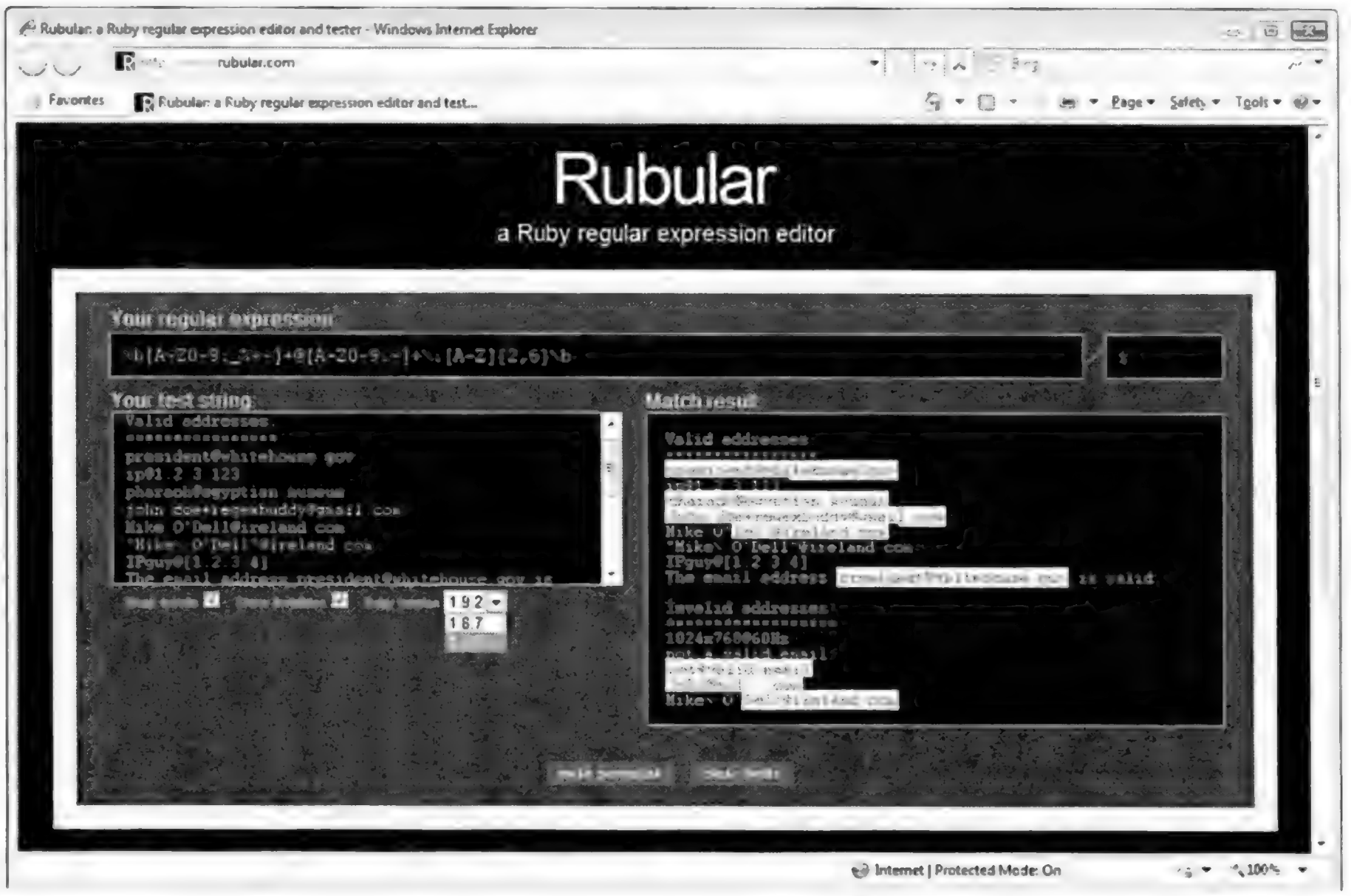


Рис. 1.7. *Rubular*

myregexp.com

Сергеем Евдокимовым (Sergey Evdokimov) было создано несколько приложений для разработчиков на языке Java, позволяющих тестировать регулярные выражения. На домашней странице *http://www.myregexp.com* (рис. 1.8) предоставляется доступ к веб-приложению тестирования регулярных выражений. Это Java-апплет, который выполняется браузером. При этом необходимо, чтобы на компьютере клиента была установлена среда выполнения Java 4 (или более поздней версии). Для оценки регулярных выражений используется пакет `java.util.regex`, который впервые появился в версии Java 4. Под диалектом регулярных выражений «Java» подразумевается именно этот пакет.

Регулярное выражение вводится в поле Regular Expression (Регулярное выражение). Желаемые параметры регулярного выражения устанавливаются с помощью меню Flags (Флаги). Кроме того, три параметра имеют собственные отдельные флажки.

Если необходимо проверить регулярное выражение, уже имеющееся в исходном программном коде на языке Java, достаточно просто скопировать строку целиком в буфер обмена, затем на странице *myregexp.com* щелкнуть на пункте меню Edit (Правка) и выбрать пункт Paste Regex from Java String (Вставить регулярное выражение из строки на языке Java). По завершении редактирования регулярного выражения в том же самом меню можно выбрать пункт Copy Regex for Java Source (Копировать

регулярное выражение в формате строки на языке Java). В меню Edit (Правка) имеются похожие команды для JavaScript и XML.

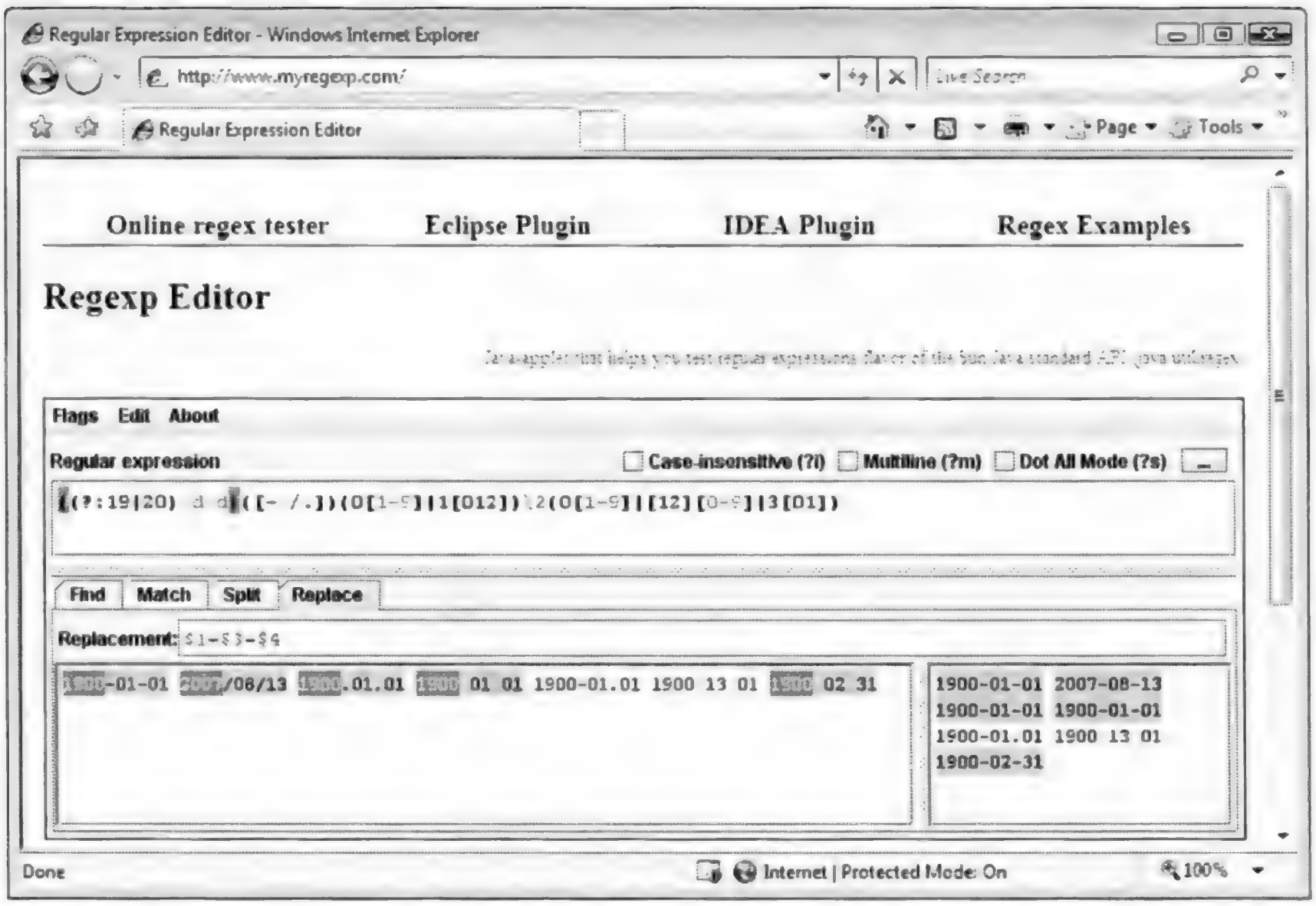


Рис. 1.8. myregexp.com

Под полем ввода регулярного выражения имеются четыре вкладки, позволяющие производить разные виды испытаний:

Find (Поиск)

Выделяет все соответствия регулярному выражению, обнаруженные в испытуемом тексте. Эти соответствия отыскиваются с помощью метода `Matcher.find()` языка Java.

Match (Соответствие)

Проверяет, соответствует ли весь испытуемый текст целиком указанному регулярному выражению. Если соответствует, выделяется весь текст целиком. Этот тест демонстрирует работу методов `String.matches()` и `Matcher.matches()`.

Split (Разбиение)

Второе поле справа отображает массив строк, возвращаемых методом `String.split()` или `Pattern.split()` при использовании указанных вами регулярного выражения и испытуемого текста.

Replace (Замена)

После ввода замещающего текста в поле справа будет выведен текст, возвращаемый методом `String.replaceAll()` или `Matcher.replaceAll()`.

Другие приложения тестирования регулярных выражений, созданные Сергеем, можно отыскать с помощью ссылок, расположенных в верхней части страницы *http://www.myregexp.com*. Одно из них является модулем расширения для среды программирования Eclipse.

Примеры настольных приложений тестирования регулярных выражений

Expresso

Expresso (не следует путать с кофе espresso (эспрессо), богатым кофеином) – это приложение для платформы .NET, позволяющее создавать и тестировать регулярные выражения. Загрузить его можно по адресу *http://www.ultrapico.com/Expresso.htm*. Для его работы на компьютере должна быть установлена платформа .NET Framework 2.0 или более поздней версии.

Для свободной загрузки доступна 60-дневная оценочная версия. По окончании оценочного периода необходимо зарегистрироваться, в противном случае программа Expresso перестанет работать. Регистрация выполняется бесплатно, но требует указать парням из Ultrapico свой адрес электронной почты. Регистрационный ключ высылается по электронной почте.

Внешний вид приложения Expresso приводится на рис. 1.9. Поле Regular Expression (Регулярное выражение), предназначенное для ввода регулярных выражений, постоянно находится в зоне видимости. Подсветка синтаксиса отсутствует. В поле Regex Analyzer (Анализатор регулярного выражения) автоматически создается краткий отчет о регулярном выражении на английском языке. Это поле также отображается постоянно.

В нижней части вкладки Design Mode (Режим проектирования) имеется возможность устанавливать такие параметры поиска совпадений, как Ignore Case (Игнорировать регистр символов). Большую часть экранного пространства занимает линия вкладок, где можно выбирать элементы для добавления в регулярное выражение. При наличии двух мониторов или одного большого можно щелчком мыши на кнопке Undock (Отделить) отделить линию вкладок, преобразовав ее в плавающую панель. Кроме того, регулярные выражения можно также создавать во вкладке Test Mode (Режим тестирования).

Во вкладке Test Mode (Режим тестирования) следует ввести или вставить испытуемый текст в левый нижний угол и щелкнуть на кнопке Run Match (Выполнить поиск соответствий), чтобы получить в поле Search Results (Результаты поиска) полный список всех соответствий. В испытуемом тексте ничего не выделяется. Чтобы выделить совпадение в испытуемом тексте, следует щелкнуть на совпадении в наборе результатов.

Во вкладке Expression Library (Библиотека выражений) выводится список примеров регулярных выражений и список последних регулярных

выражений. Испытуемое регулярное выражение добавляется в этот список всякий раз, когда выполняется щелчок на кнопке Run Match (Выполнить поиск соответствий). Библиотека допускает возможность редактирования посредством меню Library (Библиотека) в главном меню программы.

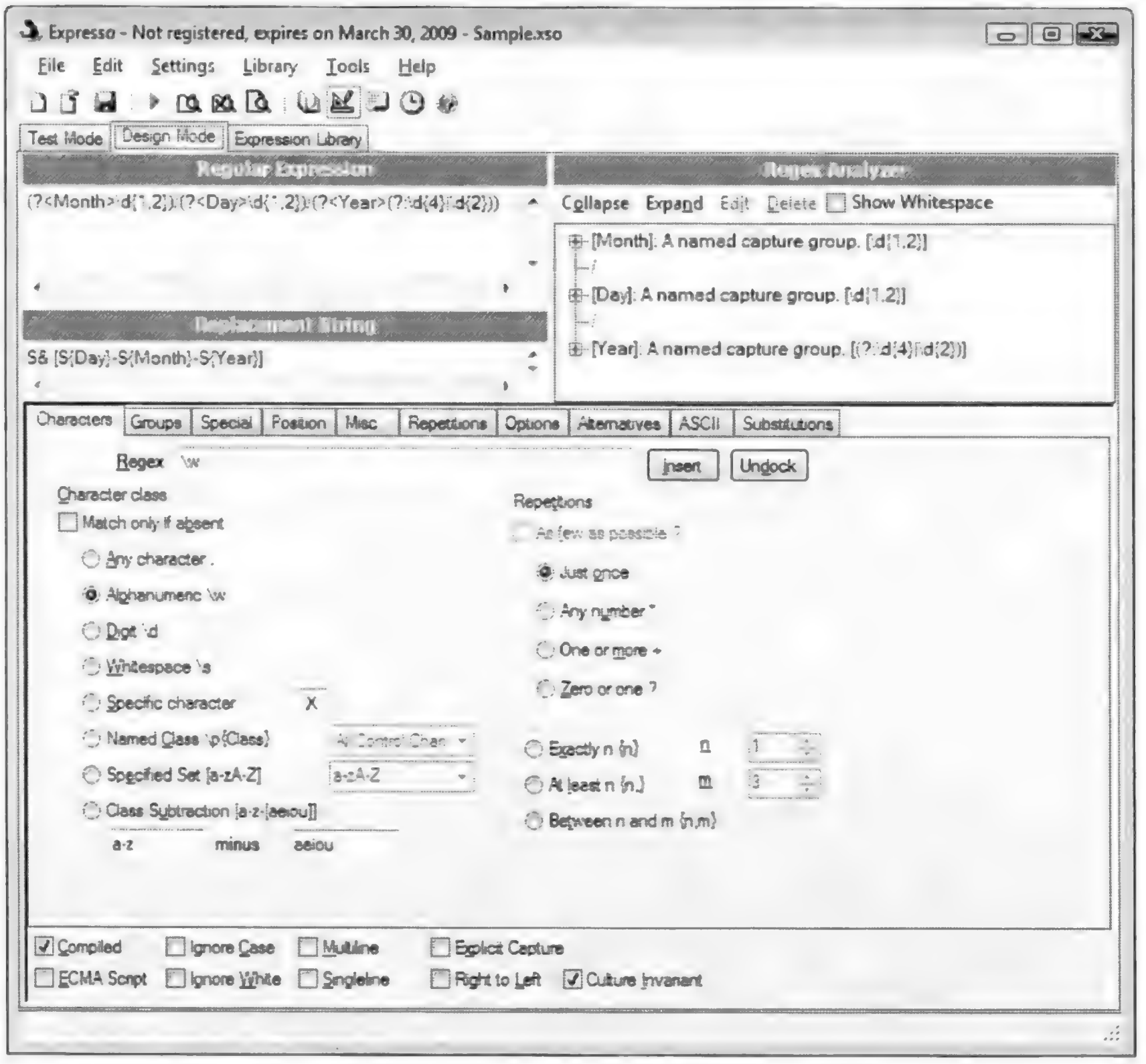


Рис. 1.9. Expresso

Regulator

Приложение Regulator, которое можно загрузить по адресу *http://sourceforge.net/projects/regulator/*, – это еще одно приложение на платформе .NET, предназначенное для создания и тестирования регулярных выражений. Последняя версия приложения требует наличия установленной версии .NET 2.0 или более поздней. Кроме того, доступны для загрузки более старые версии приложения для .NET 1.x. Regulator – это открытое программное обеспечение, для приобретения которого не требуется платить деньги или проходить процедуру регистрации.

Вся функциональность приложения Regulator доступна на единственном экране (рис. 1.10). Вкладка New Document (Новый документ) – это место, где вводится регулярное выражение. Подсветка синтаксиса выполняется автоматически, но синтаксические ошибки, присутствующие в регулярном выражении, не выделяются. Щелчком правой кнопки мыши вызывается контекстное меню, в котором можно выбирать добавляемые элементы регулярного выражения. Параметры регулярного выражения можно устанавливать с помощью кнопок на главной панели инструментов. Ярлыки на кнопках могут ввести в заблуждение, поэтому дожидайтесь появления всплывающей подсказки, чтобы определить назначение конкретной кнопки.

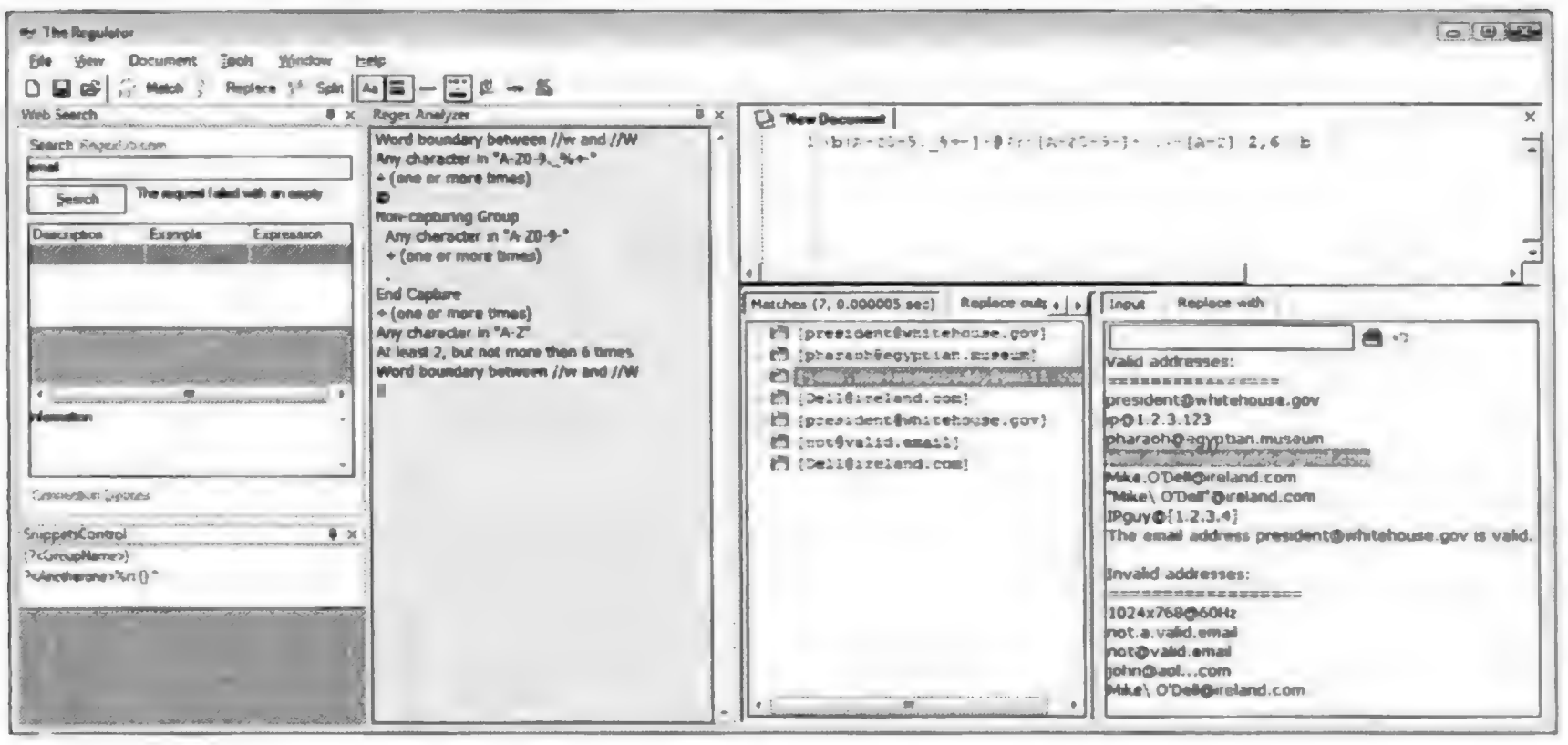


Рис. 1.10. *Regulator*

Ниже и правее области ввода регулярного выражения имеется кнопка Input (Ввод), щелчок на которой приводит к появлению области, куда можно вставить испытуемый текст. Щелчок на кнопке Replace with (Заменить на) открывает область ввода замещающего текста, используемого в операции поиска с заменой. Ниже и левее области ввода регулярного выражения выводятся результаты работы регулярного выражения. Результат не обновляется автоматически – чтобы обновить результат, необходимо щелкнуть на кнопке Match (Сопоставить), Replace (Заменить) или Split (Разбить) в панели инструментов. Фрагменты в испытуемом тексте не выделяются. Щелчок на совпадении в списке результатов приводит к выделению соответствующего фрагмента в испытуемом тексте.

В панели Regex Analyzer (Анализатор регулярного выражения) выводится простой отчет о регулярном выражении на английском языке, но этот отчет не обновляется автоматически и не является интерактивным. Чтобы обновить отчет, следует выбрать пункт Regex Analyzer (Анализатор регулярного выражения) в меню View (Вид), даже если панель анализатора видима. Щелчок мышью в панели анализатора приводит лишь к перемещению текстового курсора.

SDL Regex Fuzzer

Из-за замысловатого названия SDL Regex Fuzzer сложно догадаться о назначении этого приложения. Компания Microsoft позиционирует его как «инструмент тестирования регулярных выражений на наличие уязвимостей для атак типа «отказ в обслуживании». Его можно загрузить бесплатно по адресу *http://www.microsoft.com/en-us/download/details.aspx?id=20095*. Для работы приложения требуется наличие установленной на компьютере версии .NET 3.5.

В действительности, приложение SDL Regex Fuzzer проверяет существование испытуемой строки, сопоставление с которой может вызвать экспоненциальный рост времени выполнения регулярного выражения. В нашей книге мы называем это *катастрофическими возвратами* (*catastrophic backtracking*). Подробнее об этой проблеме рассказывается в рецепте 2.15, где также будут представлены возможные решения. Регулярное выражение, подверженное проблеме катастрофических возвратов, может вызвать зависание или аварийное завершение приложения. Если приложение выполняется на сервере, эту уязвимость можно использовать для атак типа «отказ в обслуживании» (denial-of-service, DOS).

На рис. 1.11 представлены результаты тестирования в приложении SDL Regex Fuzzer. В разделе Step 1 (Шаг 1) мы вставили регулярное выражение из рецепта 2.15. Так как это выражение никогда не совпадет с неASCII символами, нет необходимости выбирать параметр в разделе Step 2 (Шаг 2). В противном случае нам пришлось бы сделать это. В разделе Step 3 (Шаг 3) мы оставили количество итераций по умолчанию (100). Спустя примерно пять секунд после щелчка на кнопке Start (Пуск) в разделе Step 4 (Шаг 4) приложение SDL Regex Fuzzer выведет пример строки, которая вызовет ошибку при сопоставлении в .NET 3.5.

К сожалению, практическая польза этого инструмента существенно ограничена из-за поддержки лишь небольшого подмножества синтаксиса регулярных выражений .NET. Когда мы попытались проверить упрощенное решение из рецепта 2.15, которое определенно было бы забраковано этим тестом, мы получили сообщение об ошибке, как показано на рис. 1.12. Надежное понимание концепций, обсуждаемых в рецепте 2.15, по-прежнему остается единственной гарантией, что вы не поставите подножку своим приложениям чрезмерно сложными регулярными выражениями.

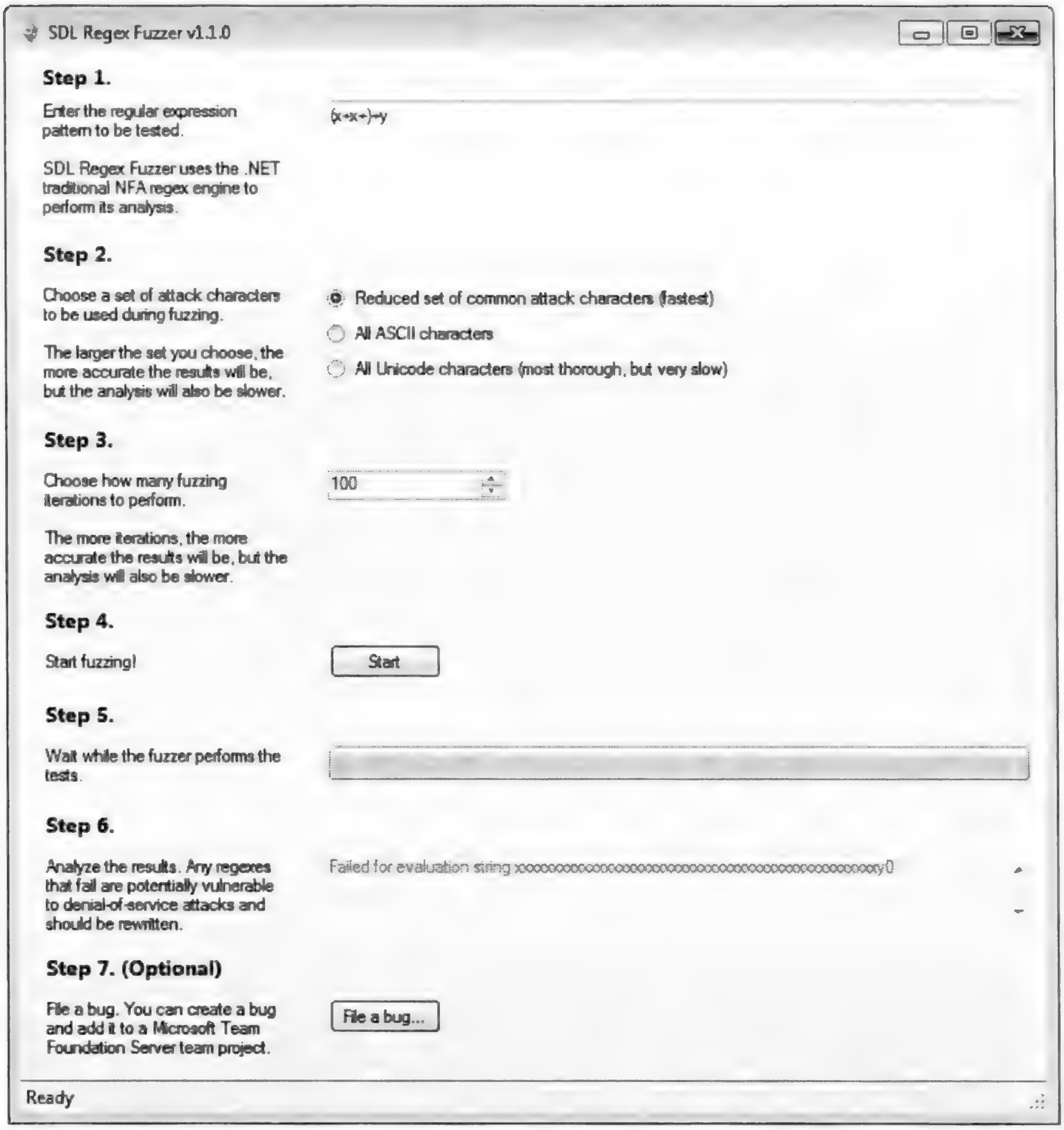


Рис. 1.11. SDL Regex Fuzzer

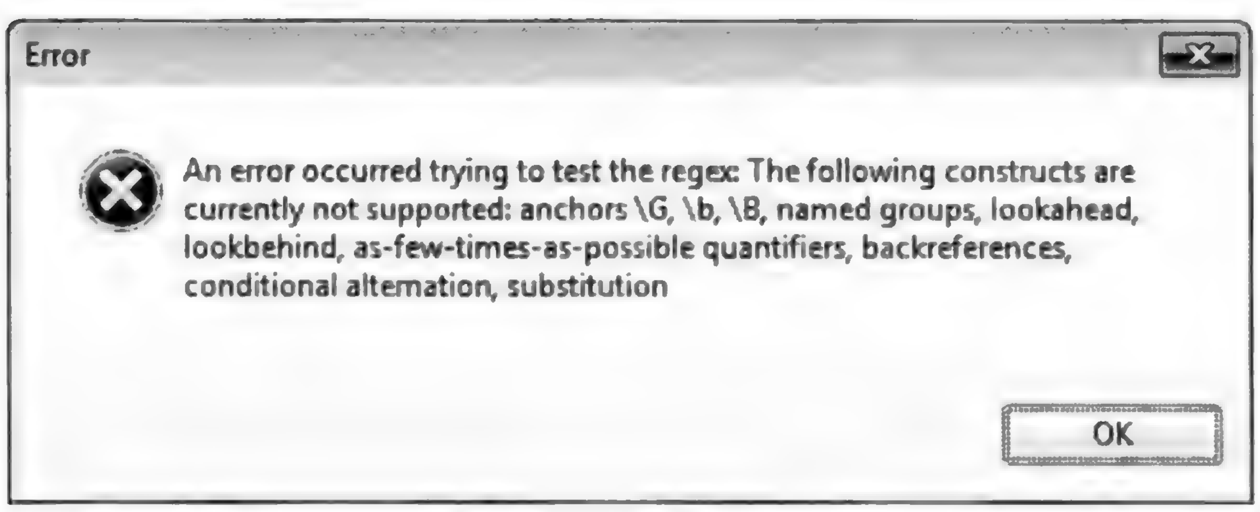


Рис. 1.12. Ограничения в SDL Regex Fuzzer

grep

Название *grep* происходит от команды `g/re/p`, которая выполняет поиск по регулярному выражению в текстовом редакторе *ed* в системе UNIX, одном из первых приложений, обеспечивших поддержку регулярных выражений. Эта команда оказалась настолько востребованной во всех системах UNIX, что была выделена в отдельную утилиту *grep*, обеспечивающую поиск в файлах с использованием регулярных выражений. Если вы используете систему UNIX, Linux или OS X, введите команду `man grep` в терминале, чтобы получить подробное описание этой утилиты.

Следующие три инструмента являются приложениями Windows, которые работают точно так же, как и утилита *grep*, и обладают дополнительными возможностями.

PowerGREP

Программа PowerGREP была разработана Яном Гойвертсом (Jan Goyvaerts), одним из авторов этой книги. Это, пожалуй, одна из наиболее мощных реализаций утилиты *grep* для платформы Windows (рис. 1.13). Программа PowerGREP использует собственный диалект регулярных выражений, соединяющий в себе лучшие черты диалектов, рассматриваемых в этой книге. Этот диалект в программе RegexBuddy помечен как «JGsoft».

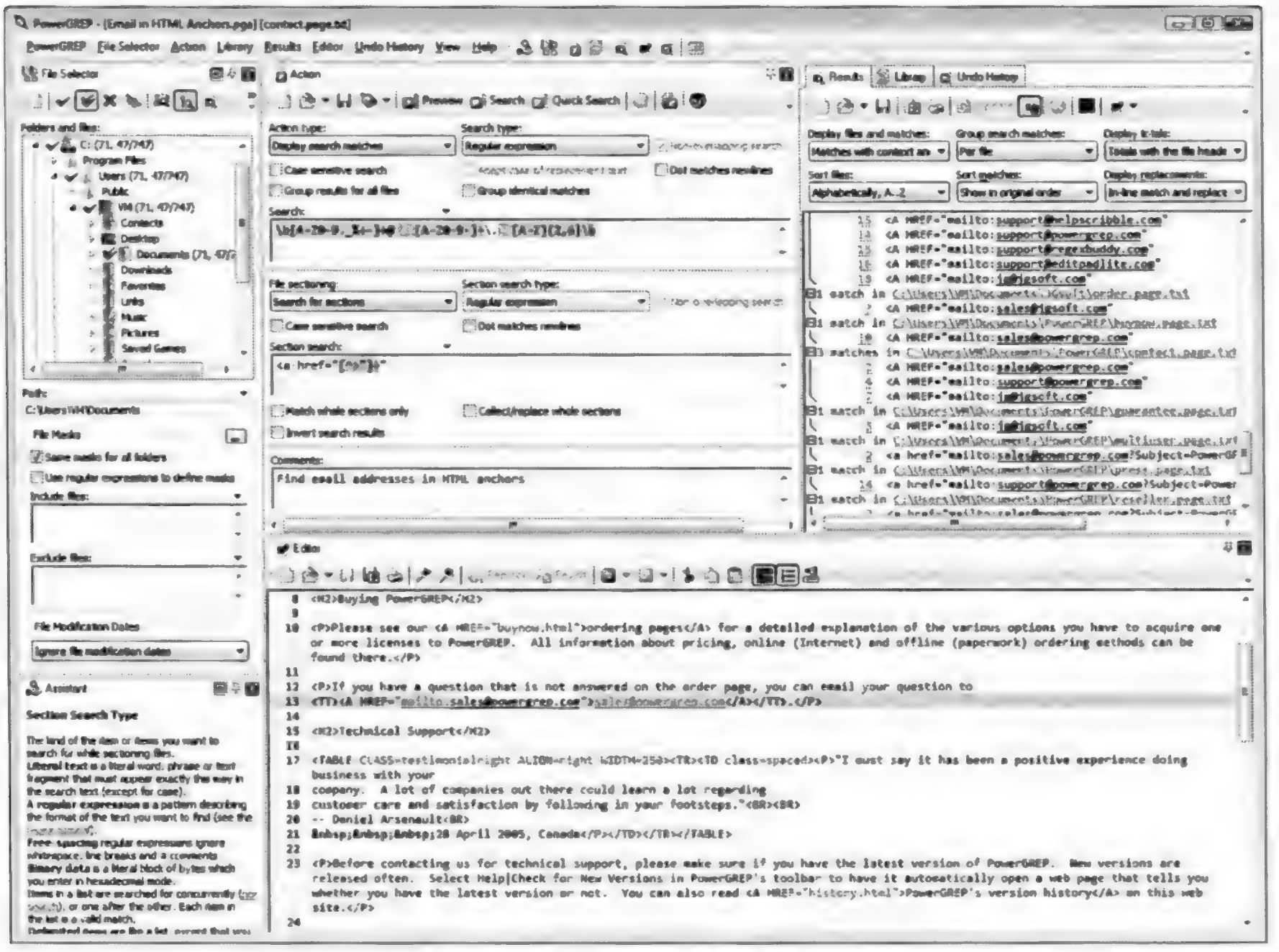


Рис. 1.13. PowerGREP

Чтобы приступить к поиску с применением регулярного выражения, достаточно просто выбрать пункт Clear (Очистить) в меню Action (Операция) и ввести регулярное выражение в поле Search (Найти), находящееся в панели Action (Операция). В панели File Selector (Выбор файла) щелчком мыши следует выбрать требуемый каталог, а в меню File Selector (Выбор файла) выбрать пункт Include File or Folder (Включить файл или папку) или Include Folder and Subfolders (Включить папку и подпапки). После этого можно выбрать пункт Execute (Выполнить) в меню Action (Операция) и тем самым запустить процедуру поиска.

Чтобы запустить поиск с заменой, после очистки операции следует выбрать в раскрывающемся списке Action type (Тип операции), находящемся в левом верхнем углу панели Action (Операция), пункт Search-and-Replace (Поиск с заменой). После этого ниже поля Search (Найти) появится поле ввода Replace (Заменить). В это поле следует ввести замещающий текст. Все остальные шаги совпадают с выполнением операции поиска.

Программа PowerGREP обладает уникальной возможностью одновременно использовать до пяти списков регулярных выражений, причем в каждом из списков может находиться произвольное число регулярных выражений. В предыдущих двух абзацах описывается, как выполняются простые операции поиска, которые могут быть выполнены с помощью любой утилиты *grep*, однако чтобы оценить весь потенциал PowerGREP, придется заняться чтением обширной документации к этой программе.

Программа PowerGREP работает в системах Windows 98, ME, 2000, XP, Vista, 7 и 8. Вы можете бесплатно загрузить оценочную версию программы по адресу *http://www.powergrep.com/PowerGREPCookbook.exe*. За исключением возможности сохранять результаты и библиотеки, оценочная версия обладает полной функциональностью в течение 15 дней фактического использования. Но хотя оценочная версия не позволяет сохранять результаты, отображаемые в панели Results (Результаты), тем не менее она будет изменять содержимое файлов при выполнении операции поиска с заменой, как и полнофункциональная версия.

Windows Grep

Программа Windows Grep (*http://www.wingrep.com*) – один из старейших *grep*-инструментов для Windows. Возраст программы проявляется в ее пользовательском интерфейсе (рис. 1.14), но она прекрасно справляется со всем, что заявлено. Она поддерживает ограниченный диалект регулярных выражений под названием POSIX ERE. Следует отметить, что этот диалект поддерживает тот же синтаксис, что и диалекты, рассматриваемые в этой книге. Программа Windows Grep относится к категории условно бесплатного программного обеспечения (shareware), то есть вы можете бесплатно загрузить ее, но предполагается, что вы заплатите за нее, если пожелаете ею пользоваться.

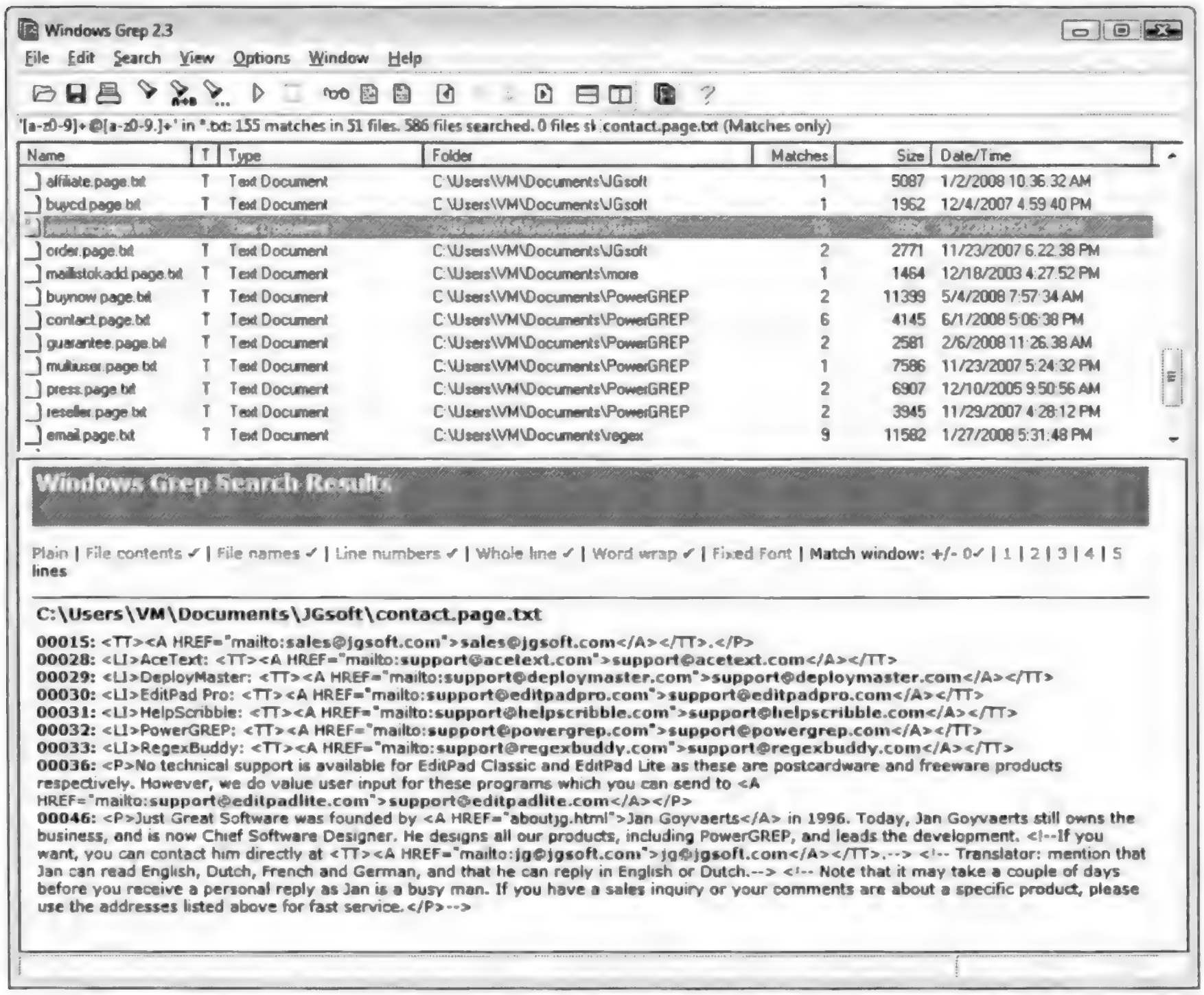


Рис. 1.14. *Windows Grep*

Чтобы выполнить поиск, следует выбрать пункт Search (Найти) в меню Search (Найти). Содержимое окна программы отличается в зависимости от выбранного в меню Options (Параметры) режима: Beginner Mode (Режим новичка) или Expert Mode (Режим эксперта). Для новичков предоставляется пошаговый мастер, а для экспертов – диалог с вкладками.

По окончании подготовительных операций программа Windows Grep немедленно приступает к поиску и представляет список файлов, в которых были обнаружены совпадения. Если щелкнуть на файле в списке, в нижней панели можно увидеть найденные в нем соответствия, а двойной щелчок открывает файл. Чтобы нижняя панель была видна постоянно, следует выбрать пункт All Matches (Все соответствия) в меню View (Вид).

Чтобы произвести поиск с заменой, в меню Search (Найти) следует выбрать пункт Replace (Заменить).

RegexRenamer

Программа RegexRenamer (рис. 1.15) в действительности не является *grep*-инструментом. Она не выполняет поиск по содержимому файлов, вместо этого она отыскивает и изменяет имена файлов. Программу можно загрузить по адресу *http://regexrenamer.sourceforge.net*. RegexRena-

mer требует наличия установленной версии 2.0 или выше платформы Microsoft .NET Framework.

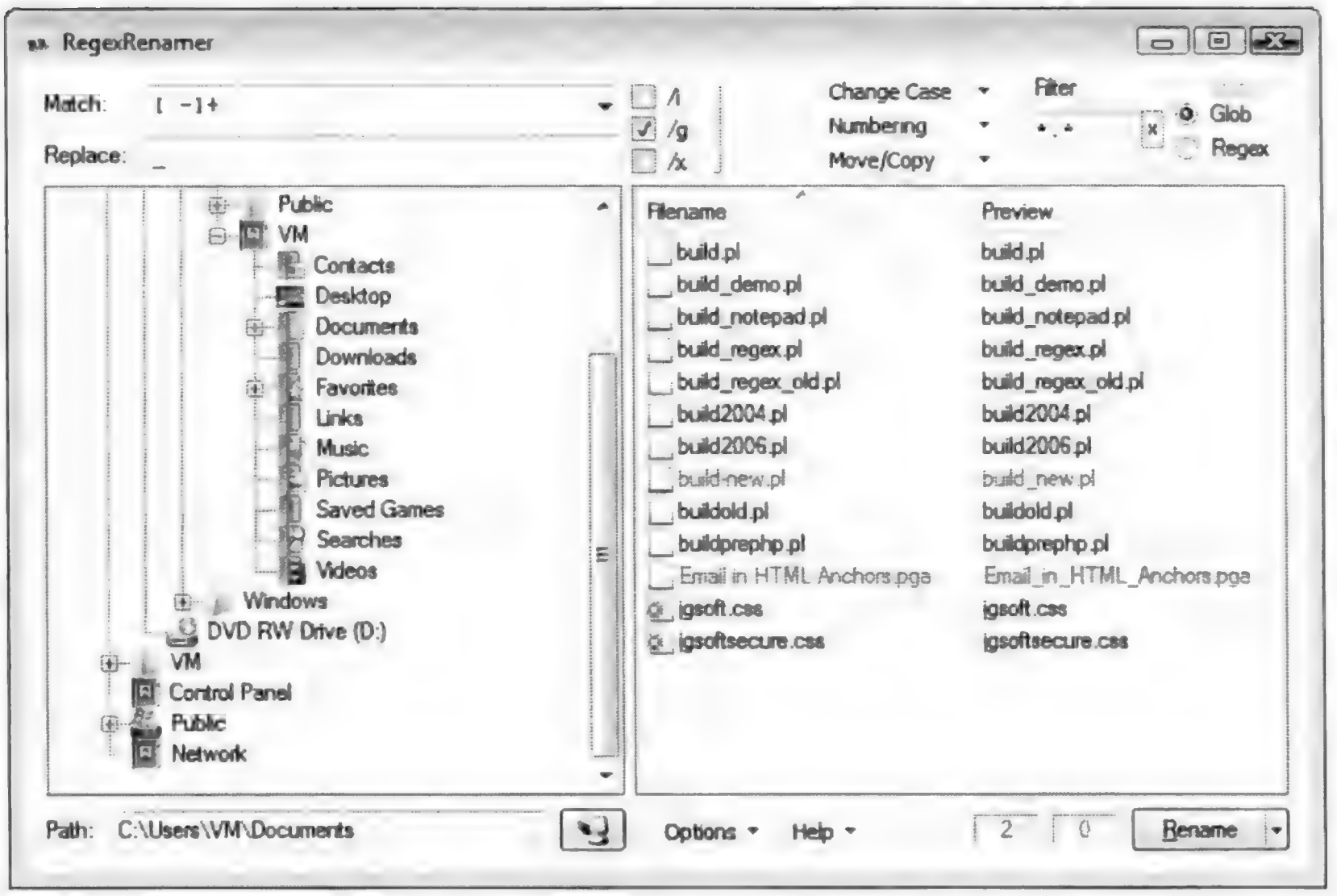


Рис. 1.15. *RegexRenamer*

Регулярное выражение вводится в поле Match (Соответствие), а замещающий текст – в поле Replace (Заменить). Чтобы включить режим нечувствительности к регистру символов, следует установить флажок /i, а при включенном флажке /g производится замена не только первого, но и всех остальных найденных соответствий в каждом имени файла. Флажок /x включает режим игнорирования дополнительных пробелов, что само по себе малополезно, так как в поле регулярного выражения можно ввести только одну строку.

Дерево каталогов слева используется для выбора папки с файлами, которые требуется переименовать. В правом верхнем углу можно определить маску для файлов или регулярное выражение фильтра. Это позволяет ограничить перечень файлов, к именам которых будет применяться операция поиска с заменой. Возможность использования одного регулярного выражения для фильтрации, а другого для выполнения операции поиска с заменой – преимущество по сравнению с попыткой решения обеих задач с помощью единственного регулярного выражения.

Популярные текстовые редакторы

Большинство современных текстовых редакторов обладает, по меньшей мере, самой примитивной поддержкой регулярных выражений. В диалоге поиска или поиска с заменой обычно присутствует флажок,

включающий режим использования регулярного выражения. Некоторые редакторы, такие как EditPad Pro, также используют регулярные выражения для обработки текста, например для организации подсветки синтаксиса или создания списков классов и функций. Все эти особенности редакторов описываются в сопроводительной документации. Ниже приводится список некоторых текстовых редакторов, обладающих поддержкой регулярных выражений:

- BBEdit (PCRE)
- Boxer Text Editor (PCRE)
- Dreamweaver (JavaScript)
- EditPad Pro (собственный диалект, вобравший в себя лучшие черты диалектов, рассматриваемых в этой книге. В программе RegexBuddy именуется «JGsoft»)
- Multi-Edit (PCRE, если выбран параметр Perl)
- Nisus Writer Pro (Ruby 1.9 [Oniguruma])
- Notepad++ (PCRE)
- NoteTab (PCRE)
- UltraEdit (PCRE)
- TextMate (Ruby 1.9 [Oniguruma])

2

Основные навыки владения регулярными выражениями

Задачи, представленные в этой главе, не являются реальными задачами, которые будут ставить перед вами босс или ваши заказчики. Это, скорее, технические задачи, с которыми вы будете сталкиваться при создании регулярных выражений, предназначенных для решения практических задач. Например, в первом рецепте объясняется, как с помощью регулярного выражения отыскать определенный текст и как обрабатывать символы, имеющие специальное значение в регулярных выражениях. Само по себе это не может быть целью, потому что нет никакой необходимости использовать регулярные выражения, когда требуется отыскать определенный текст. Однако при создании регулярных выражений вам наверняка потребуется отыскивать соответствия с определенным текстом, поэтому необходимо знать, какие символы следует экранировать, а как раз об этом и рассказывается в рецепте 2.1.

Первыми приводятся рецепты, описывающие очень простые приемы использования регулярных выражений. Если у вас уже имеется некоторый опыт работы с регулярными выражениями, вы можете просто бегло просмотреть их или вообще пропустить. Последующие рецепты в этой главе определенно будут содержать новую для вас информацию, если, конечно, вы не прочитали от корки до корки книгу «Mastering Regular Expressions» Джеффри Фридла (Jeffrey E. F. Friedl), выпущенную издательством O'Reilly[1].

Мы придумывали рецепты для этой главы так, чтобы каждый из них описывал один аспект синтаксиса регулярных выражений. Все вместе они образуют законченный учебник по регулярным выражениям.

---

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

Рекомендуем прочитать его от начала до конца, чтобы твердо усвоить основы регулярных выражений. Однако можно сразу перейти к регулярным выражениям, имеющим практическую ценность, которые приводятся в главах с 4 по 9, и возвращаться по ссылкам к этой главе при встрече с незнакомыми синтаксическими конструкциями.

В этой вводной главе рассматриваются только регулярные выражения и полностью исключаются из рассмотрения какие-либо вопросы, связанные с программированием. Следующая глава наполнена листингами программного кода. Вы можете заглянуть в раздел «Языки программирования и диалекты регулярных выражений» главы 3, чтобы узнать, какой диалект регулярных выражений используется в вашем языке программирования. Сами диалекты, о которых будет говориться в этой главе, были представлены в главе 1 в разделе «Диалекты регулярных выражений, рассматриваемые в этой книге».

2.1. Соответствие литеральному тексту

Задача

Создать регулярное выражение, в точности соответствующее следующему восхитительному предложению: `The punctuation characters in the ASCII table are: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`.

Цель этого рецепта – познакомить вас с символами, имеющими специальное значение в регулярных выражениях, и с символами, которые всегда соответствуют самим себе.

Решение

Предложению, указанному выше, соответствует следующее регулярное выражение:

```
The•punctuation•characters•in•the•ASCII•table•are:•↵
!"#\$%&'\(\)\*\+,-\./:;<=>\?@\[\\]\^_`\{\|}~
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Любое регулярное выражение, в котором символы `$()*+.?[\^{|` отсутствуют, просто соответствует само себе. Например, чтобы отыскать в тексте фразу `Mary had a little lamb`, достаточно воспользоваться регулярным выражением `‹Mary•had•a•little•lamb›`. При этом совершенно неважно, установлен ли флажок Regular Expression (Регулярное выражение) в диалоге поиска текстового редактора.

Двенадцать знаков препинания, которые обеспечивают всю магическую силу регулярных выражений, называются *метасимволами*. Если необходимо, чтобы регулярное выражение находило соответствие с этими

символами буквально, их следует экранировать, помещая перед ними символ обратного слэша. То есть регулярному выражению ‹\$\(\)*\+\.\?\[\\\^\{\|› соответствует текст $()*+.?[\^{|.

Обратите внимание, что в списке отсутствуют: закрывающая квадратная скобка], дефис - и закрывающая фигурная скобка }. Первые два символа интерпретируются как метасимволы, только если они стоят после неэкранированной открывающей квадратной скобки [, а } – только после неэкранированного символа {. Нет никакой необходимости экранировать символ }. Правила использования метасимволов в блоках, заключенных между символами [и], описываются в рецепте 2.3.

Экранирование любых других не алфавитно-цифровых символов не влияет на работу регулярного выражения; по крайней мере, это утверждение справедливо для всех диалектов, рассматриваемых в книге. Экранирование алфавитно-цифровых символов либо может придавать им особое значение, либо вызывать синтаксическую ошибку.

Те, кто плохо знаком с регулярными выражениями, часто стремятся экранировать каждый знак препинания, попадающий в поле зрения. Не старайтесь показать всем, что вы – новичок. Разумно используйте экранирование. Частокол ненужных символов обратного слэша осложняет чтение регулярного выражения, особенно когда все эти символы обратного слэша требуется удваивать, чтобы представить регулярное выражение как строковый литерал в исходном тексте программы.

Варианты

Экранирование блока

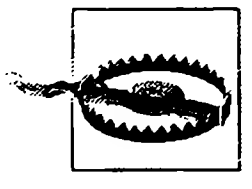
В диалектах, поддерживающих прием *экранирования блока*, есть возможность сделать решение более читаемым:

```
The•punctuation•characters•in•the•ASCII•table•are:•↵
\Q!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~\E
```

Параметры: нет
Диалекты: Java 6, PCRE, Perl

Диалекты Perl, PCRE и Java поддерживают конструкции ‹\Q› и ‹\E›. Конструкция ‹\Q› подавляет интерпретацию всех метасимволов, включая и символ обратного слэша, пока не встретится конструкция ‹\E›. Если опустить конструкцию ‹\E›, все символы, находящиеся в регулярном выражении после конструкции ‹\Q›, будут интерпретироваться буквально.

Единственное преимущество выражения ‹\Q...\E› состоит в том, что его проще читать, чем ‹\.\.\.›.



Хотя версии Java 4 и 5 поддерживают эту особенность, тем не менее лучше ею не пользоваться. Ошибки в реализации приводят к тому, что регулярному выражению с конструкцией ‹\Q···\E› соответствует совсем не то, что соответствует этому же регулярному выражению

в диалектах PCRE, Perl и Java 6. Эти ошибки были исправлены в версии Java 6, благодаря чему этот диалект дает тот же результат, что и диалекты PCRE и Perl.

Сопоставление без учета регистра символов

По умолчанию регулярные выражения чувствительны к регистру символов. Выражению ‹regex› соответствует текст regex, но не Regex, REGEX или ReGeX. Чтобы регулярному выражению ‹regex› соответствовали все эти варианты, необходимо включить режим нечувствительности к регистру символов.

В большинстве приложений для этого достаточно установить или сбросить флажок. Все языки программирования, обсуждаемые в следующей главе, имеют флаг или свойство, с помощью которого можно управлять режимом чувствительности регулярного выражения к регистру символов. В рецепте 3.4, в следующей главе, описывается, как в программном коде применять параметры регулярного выражения, соответствующие приводимому в книге регулярному выражению.

```
ascii
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если режим нечувствительности к регистру символов не был включен за пределами регулярного выражения, это можно сделать с помощью модификатора режима ‹(?i)›, например ‹(?i)regex›. Этот прием работает в диалектах .NET, Java, PCRE, Perl, Python и Ruby. При использовании библиотеки XRegExp его можно применять и в JavaScript.

```
(?i)ascii
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Диалекты .NET, Java, PCRE, Perl и Ruby поддерживают локальные модификаторы режима, которые воздействуют только на часть регулярного выражения. Например, регулярному выражению ‹sensitive(?i)caseless(?-i)sensitive› соответствует текст sensitiveCASELESSsensitive, но не SENSITIVEcaselessSENSITIVE. Модификатор ‹(?i)› включает режим нечувствительности к регистру символов до конца регулярного выражения, а модификатор ‹(?-i)› отключает его до конца регулярного выражения. Они действуют как простые переключатели.

В рецепте 2.9 показано, как использовать локальные модификаторы режима с группами.

См. также

Рецепт 2.3 описывает символьные классы. Внутри символьных классов используются иные метасимволы, чем за их пределами.

Рецепт 5.14 демонстрирует, как экранировать все метасимволы в строке при встраивании ее в регулярное выражение в качестве литерала. Для этого строка преобразуется в регулярное выражение, соответствующее строке буквально.

В разделе «Пример решения на языке JavaScript» в рецепте 5.2 демонстрируется пример кода на JavaScript, экранирующий все метасимволы регулярных выражений. В некоторых языках программирования для этих целей существуют встроенные команды.

2.2. Соответствие непечатным символам

Задача

Написать регулярное выражение, которому соответствовала бы строка, состоящая из следующих управляющих символов ASCII: bell, escape, form feed, line feed, carriage return, horizontal tab, vertical tab. Эти символы имеют следующие шестнадцатеричные коды ASCII: 07, 1B, 0C, 0A, 0D, 09, 0B.

Этот рецепт демонстрирует, как использовать экранированные последовательности и ссылаться на символы по их шестнадцатеричным кодам.

Решение

```
\a\e\f\n\r\t\v
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Python, Ruby

```
\x07\x1B\f\n\r\t\v
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python, Ruby

```
\a\e\f\n\r\t\x0B
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Для представления семи наиболее часто используемых управляющих символов ASCII имеются специальные *экранированные последовательности*. Все они состоят из символа обратного слэша и следующей за ним буквы. Это тот же самый синтаксис, что используется во многих языках программирования для представления управляющих символов в строковых литералах. В табл. 2.1 приводится перечень наиболее часто используемых непечатных символов и их представления.

В версиях Perl 5.10 и выше и в PCRE 7.2 и выше не поддерживается экранированная последовательность ‹\v› (вертикальная табуляция). В этих диалектах ‹\v› соответствует всем вертикальным пробельным символам,

в число которых входят: вертикальная табуляция, символы разрыва строк и разделители Юникода строк и абзацев. По этой причине в Perl и PCRE следует использовать иной синтаксис определения вертикальной табуляции.

Таблица 2.1. Непечатные символы

Представление	Значение	Шестнадцатеричное представление	Диалекты
‹\a›	bell	0x07	.NET, Java, PCRE, Perl, Python, Ruby
‹\e›	escape	0x1B	.NET, Java, PCRE, Perl, Ruby
‹\f›	form feed	0x0C	.NET, Java, JavaScript, PCRE, Perl, Python, Ruby
‹\n›	line feed (новая строка)	0x0A	.NET, Java, JavaScript, PCRE, Perl, Python, Ruby
‹\r›	carriage return	0x0D	.NET, Java, JavaScript, PCRE, Perl, Python, Ruby
‹\t›	horizontal tab	0x09	.NET, Java, JavaScript, PCRE, Perl, Python, Ruby
‹\v›	vertical tab	0x0B	.NET, Java, JavaScript, Python, Ruby

Диалект JavaScript не поддерживает последовательности ‹\a› и ‹\e›. Поэтому в JavaScript также необходимо использовать другое решение.

Эти управляющие символы и альтернативный синтаксис, который демонстрируется в следующем разделе, с одинаковым успехом могут использоваться в регулярных выражениях как внутри, так и за пределами символьных классов.

Варианты представления непечатных символов

26 управляющих символов

Ниже приводится другой способ представления в регулярных выражениях этих семи управляющих символов ASCII:

```
\cG\x1B\cL\cJ\cM\cI\cK
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Ruby 1.9

С помощью последовательностей от ‹\cA› до ‹\cZ› определяется соответствие с любым из 26 управляющих символов, занимающим позицию от 1 до 26 в таблице символов ASCII. Символ c должен указываться в нижнем регистре. Буква, следующая за ним, в большинстве диалектов может указываться в любом регистре, но мы рекомендуем всегда исполь-

зовать символы верхнего регистра. В диалекте Java это обязательное условие.

Этот синтаксис может оказаться удобным, если вы привыкли вводить управляющие символы в консоли, нажимая клавишу Ctrl одновременно с алфавитной клавишей. Комбинация клавиш Ctrl-H в терминалах соответствует символу забоя (backspace). В регулярных выражениях символу забоя соответствует последовательность ‹\cH›.

Этот синтаксис не поддерживается механизмом регулярных выражений языка Python и классическим механизмом Ruby в Ruby 1.8, но он поддерживается механизмом Onigurama в Ruby 1.9.

Управляющий символ escape, занимающий позицию 27 в таблице ASCII, не имеет эквивалента в английском алфавите, поэтому в наших регулярных выражениях он представлен последовательностью ‹\x1B›.

7-битный набор символов

Ниже приводится еще один способ сопоставления со списком из семи управляющих символов:

```
\x07\x1B\x0C\x0A\x0D\x09\x0B
```

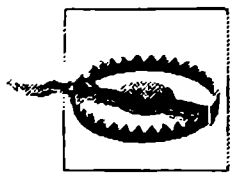
Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Последовательности из экранированного символа \x в нижнем регистре, за которым следуют две шестнадцатеричные цифры в верхнем регистре, соответствует один символ из набора ASCII. На рис. 2.1 показано соответствие шестнадцатеричных комбинаций от ‹\x00› до ‹\x7F› и символов из набора ASCII. Таблица упорядочена так, чтобы первая цифра в комбинации возрастала по направлению вниз, а вторая – вправо.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

***Рис. 2.1.** Таблица символов ASCII*

Соответствие комбинациям от ‹\x80› до ‹\xFF› зависит от того, как они интерпретируются вашим механизмом регулярных выражений и в какой кодировке набран испытуемый текст. Мы не рекомендуем использовать последовательности от ‹\x80› до ‹\xFF›. Вместо них лучше использовать кодовые пункты Юникода, описываемые в рецепте 2.7.



В случае применения Ruby 1.8 или библиотеки PCRE, скомпилированной без поддержки кодировки UTF-8, воспользоваться кодовыми пунктами Юникода будет невозможно. Ruby 1.8 и PCRE без поддержки UTF-8 представляют собой 8-битовые механизмы регулярных выражений. Они ничего не знают о кодировках символов и многобайтовых символах. Последовательность `\xAA` в этих механизмах просто соответствует байту со значением `0xAA` независимо от того, представляет ли этот байт отдельный символ с кодом `0xAA` или является частью многобайтового символа.

См. также

Рецепт 2.7, где описывается, как обеспечить совпадение регулярного выражения с конкретными символами Юникода. Если ваш механизм регулярных выражений поддерживает Юникод, этот способ можно также использовать для поиска непечатаемых символов.

2.3. Сопоставление с одним символом из нескольких

Задача

Создать регулярное выражение, которому соответствовали бы любые ошибочные написания слова `calendar`, чтобы иметь возможность отыскивать это слово в документе, не полагаясь на грамотность автора. Предполагается, что на месте любой гласной буквы может использоваться символ `a` или `e`. Создать второе регулярное выражение, которому соответствовала бы единственная шестнадцатеричная цифра. Создать третье регулярное выражение, которому соответствовал бы единственный символ, не являющийся шестнадцатеричной цифрой.

Цель этого рецепта – рассказать о такой важной и часто используемой конструкции, как *символьный класс*.

Решение

Ошибки в слове calendar

```
c[ae]l[ae]nd[ae]r
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричная цифра

```
[a-fA-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Нешестнадцатеричная цифра

```
[^a-fA-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Последовательность, заключенная в квадратные скобки, называется *символьным классом*. Символьному классу соответствует единственный символ, совпадающий с любым из перечисленных символов. В первом регулярном выражении указано три класса, каждому из которых соответствует символ a или e. Это обеспечивает свободу выбора. Когда с помощью этого регулярного выражения проверяется слово calendar, первому символьному классу соответствует символ a, второму – e и третьему – a.

Внутри символьных классов только четыре символа имеют специальное назначение: \, ^, - и]. В диалектах Java и .NET открывающая квадратная скобка [также является метасимволом внутри символьных классов.

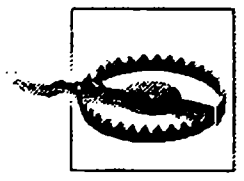
Символ обратного слэша всегда экранирует символ, следующий за ним, так же как и за пределами символьного класса. Экранированный символ может представлять единственный символ, а также начало или конец диапазона. Другие четыре метасимвола приобретают специальное назначение, только если находятся в определенной позиции. Они могут представлять литералы символов в символьных классах без дополнительного экранирования путем помещения их в позицию, где они утрачивают специальное назначение. Выражение ‹[][^-]› иллюстрирует эту особенность. Она свойственна всем диалектам регулярных выражений, рассматриваемых в книге, кроме JavaScript, строго придерживающегося стандарта. Конструкция ‹[]› в JavaScript всегда интерпретируется как пустой символьный класс, который никогда не находит совпадений. Однако мы рекомендуем всегда экранировать эти метасимволы; предыдущее регулярное выражение следовало бы записать так: ‹[\]\[\^\-]›. Экранирование метасимволов упрощает понимание регулярного выражения.

Все остальные символы интерпретируются буквально и просто добавляют себя в символьный класс. Регулярному выражению ‹[$()*+.?{|]› соответствует любой из девяти символов, перечисленных между квадратными скобками. Эти девять символов приобретают специальное значение только за пределами символьных классов. Внутри символьных классов они интерпретируются буквально. Их экранирование только ухудшит читаемость регулярного выражения.

Алфавитно-цифровые символы не должны экранироваться обратным слэшем, так как в этом случае либо может возникать ошибка, либо может создаваться специальный символ (имеющий специальное значение в регулярных выражениях). В наших пояснениях к некоторым специальным символам, например в рецепте 2.2, мы указывали, что они могут

использоваться внутри символьных классов. Все эти специальные символы состоят из слэша и символа, иногда сопровождаемых рядом других символов. Так, символьному классу ‹[\r\n]› соответствует символ возврата каретки \r или перевода строки \n.

Символ крышки ^ в символьном классе означает отрицание, если он следует непосредственно за открывающей квадратной скобкой. Он обеспечивает соответствие символьному классу любых символов, которые *отсутствуют* в списке.



Во всех диалектах, обсуждаемых в этой книге, инвертированному символьному классу соответствуют символы конца строки, если они отсутствуют в инвертированном символьном классе. Не позволяйте своим регулярным выражениям по ошибке продолжать поиск в множестве строк.

Символ дефиса (-) определяет *диапазон*, когда находится между двумя символами. Диапазон в символьном классе включает в себя символ, стоящий перед дефисом, символ, стоящий после дефиса, и все символы, расположенные между ними в порядке следования их числовых кодов. Чтобы узнать, какие числовые коды соответствуют символам, необходимо обратиться к таблице символов ASCII или Юникода. Символьный класс ‹[A-z]› включает в себя все символы, расположенные в таблице ASCII между символом верхнего регистра A и символом нижнего регистра z. Этот диапазон также включает в себя некоторые знаки препинания; так, символьному классу ‹[A-Z\[\\\]\^_`a-z]› соответствуют те же самые символы, только объявлен он более явно. Мы рекомендуем создавать диапазоны, включающие только цифры или только буквы, причем только верхнего или только нижнего регистра.

Перевернутые диапазоны, такие как ‹[z-a]›, не допускаются.

Варианты

Сокращения

Шесть специальных символов, состоящих из обратного слэша и следующего за ним алфавитного символа, представляют *сокращенную форму записи символьных классов*: ‹\d›, ‹\D›, ‹\w›, ‹\W›, ‹\s› и ‹\S›. Их можно использовать как внутри, так и за пределами символьных классов. Каждой сокращенной форме записи, включающей символ нижнего регистра, соответствует сокращенная форма записи, включающая символ верхнего регистра, имеющая противоположный смысл.

Так, последовательностям ‹\d› и ‹[\d]› соответствует один цифровой символ. Последовательности ‹\D› соответствует любой символ, *не* являющийся цифрой, а ее эквивалентом является запись ‹[^\d]›.

Вот как можно использовать сокращенную форму ‹\d›, чтобы переписать регулярное выражение, совпадающее с шестнадцатеричной цифрой, представленное выше в этом рецепте:

```
[a-fA-F\d]
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Метасимволу ‹\w› соответствует один *символ слова*. Символ слова – это символ, который может входить в состав слова. Сюда относятся алфавитные символы, цифры и символ подчеркивания. Такой набор символов может показаться странным, но он был выбран таким, потому что из этих символов обычно составляются идентификаторы в языках программирования. Метасимволу ‹\W› соответствует любой символ, который не может являться частью такого слова.

В диалектах Java (в версиях с 4 по 6), JavaScript, PCRE и Ruby метасимвол ‹\w› всегда идентичен классу ‹[a-zA-Z0-9_]›. В .NET он также включает алфавитно-цифровые символы любых других алфавитов (кириллица, тайский алфавит и прочие). В Java 7 символы других алфавитов включаются, только если установлен флаг UNICODE_CHARACTER_CLASS. В Python 2.x символы других алфавитов включаются, только если при создании регулярного выражения передавался флаг UNICODE или U. В Python 3.x символы других алфавитов включаются по умолчанию, но есть возможность заставить ‹\w› совпадать только с символами ASCII, если установить флаг ASCII или A. В Perl 5.14 флаг /a (ASCII) превращает ‹\w› в эквивалент символьного класса ‹[a-zA-Z0-9_]›, флаг /u (Unicode) включает все алфавиты, поддерживаемые стандартом Юникода, а флаг /l (locale – локаль) делает метасимвол ‹\w› зависимым от текущих национальных настроек. В версиях Perl, предшествовавших версии 5.14, а также в версии Perl 5.14 (при использовании флага /d (default – по умолчанию) или когда не используется ни один из флагов /adlu) метасимвол ‹\w› автоматически включает алфавиты Юникода, если испытуемая строка или само регулярное выражение закодировано в кодировке UTF-8, или если регулярное выражение включает коды символов со значениями выше 255, такие как ‹\x{100}›, или свойства Юникода, такие как ‹\p{L}›. В противном случае метасимвол ‹\w› соответствует только символам ASCII.

К метасимволу ‹\d› применяются те же правила во всех диалектах. В .NET цифры из других алфавитов всегда включаются в класс. В Python поведение этого метасимвола зависит от флагов UNICODE и ASCII и от версии Python – 2.x или 3.x. В Perl 5.14 его поведение зависит от флагов /adlu. В предыдущих версиях Perl оно зависит от кодировки испытуемой строки и регулярного выражения, а также от наличия каких-либо лексем Юникода.

Метасимволу ‹\s› соответствует любой *пробельный символ*. Сюда входят символы пробела, табуляции и символы конца строки. Метасимволу ‹\S› соответствует любой символ, не соответствующий метасимволу ‹\s›.

В .NET и JavaScript метасимволу ‹\s› также соответствуют все символы, определяемые стандартом Юникода как пробельные. В Java, Perl и Python метасимвол ‹\s› подчиняется тем же правилам, что и метасимволы ‹\w› и ‹\d›.

Примечательно, что в диалекте JavaScript для ‹\s› используется Юникод, а для ‹\d› и ‹\w› – ASCII. Еще одно противоречие несет в себе метасимвол ‹\b›. Метасимвол ‹\b› не является краткой формой представления символьного класса, а обозначает *границу слова*. Можно было бы ожидать, что ‹\b› поддерживает Юникод, когда эту кодировку поддерживает метасимвол ‹\w›, и только ASCII, когда ‹\w› поддерживает только ASCII, но это не всегда так. Подробнее об этом рассказывается в подразделе «Символы слов» в рецепте 2.6.

Поиск без учета регистра символов

```
(?i)[A-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
(?i)[^A-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Нечувствительность к регистру символов, определяемая с помощью внешнего флага (рецепт 3.4) или с помощью модификатора режима внутри регулярного выражения (рецепт 2.1, раздел «Сопоставление без учета регистра символов»), также оказывает воздействие и на символьные классы. Только что представленные регулярные выражения эквивалентны регулярным выражениям, представленным в первоначальном решении.

Диалект JavaScript также следует этому правилу, но он не поддерживает модификатор ‹(?i)›. Чтобы отключить чувствительность регулярного выражения к регистру символов в JavaScript, необходимо установить флаг /i при его создании или использовать библиотеку XRegExp, добавляющую поддержку модификаторов режимов в начале регулярных выражений.

Особенности, характерные для разных диалектов

Разность символьных классов в .NET

```
[a-zA-Z0-9-[g-zG-Z]]
```

Параметры: нет
Диалекты: .NET 2.0 или выше

Данному регулярному выражению соответствует одна шестнадцатеричная цифра, но реализовано оно косвенным образом. Базовому символьному классу соответствует любой алфавитно-цифровой символ,

а вложенному классу, который вычитается из базового, символы от g до z. Вложенный вычитаемый класс должен находиться в конце базового класса и предваряться символом дефиса: ‹[*class*-[*subtract*]]›.

Операцию *вычитания* символьных классов удобно использовать, в частности, при работе со свойствами, блоками и алфавитами Юникода. Например, выражению ‹\p{IsThai}› соответствует любой символ тайского алфавита. Выражению ‹\P{N}› соответствует любой символ, у которого отсутствует свойство Number. Разности этих двух классов ‹[\p{IsThai}-[\P{N}]]› соответствует любая из 10 цифр тайского алфавита. Более подробно о работе со свойствами Юникода рассказывается в рецепте 2.7.

Объединение, разность и пересечение символьных классов в Java

Диалект Java допускает вложение одного символьного класса в другой. При непосредственном включении одного символьного класса в другой получается *объединение* двух классов. Допускается создавать столько уровней вложения, сколько потребуется. Выражения ‹[a-f[A-F][0-9]]› и ‹[a-f[A-F[0-9]]]› используют объединение символьных классов. Они соответствуют шестнадцатеричным цифрам, как и первоначальное выражение, но в них отсутствуют дополнительные квадратные скобки.

Регулярное выражение ‹[\w&&[a-fA-F0-9\s]]› соответствует шестнадцатеричным цифрам, используя пересечение символьных классов. Это выражение могло бы получить приз в соревновании на самое запутанное выражение. Базовому символьному классу ‹[\w]› соответствует любой символ слова. Вложенному классу ‹[a-fA-F0-9\s]› соответствует любая шестнадцатеричная цифра и любой пробельный символ. Результатом является пересечение двух классов, которому соответствуют только шестнадцатеричные цифры и ничто иное. Так как базовому классу не соответствуют пробельные символы, а вложенному классу не соответствуют символы ‹[g-zG-Z_]›, они исключаются из окончательного символьного класса и остаются только шестнадцатеричные цифры.

Выражение ‹[a-zA-Z0-9&&[^g-zG-Z]]› использует разность символьных классов. Ему соответствует одна шестнадцатеричная цифра и тоже косвенным образом. Базовому символьному классу ‹[a-zA-Z0-9]› соответствует любой алфавитно-цифровой символ. Вложенному классу ‹[^g-zG-Z]›, который вычитается из базового, – символы от g до z. Этот вложенный класс инвертируется, и ему предшествуют два символа амперсанда: ‹[*class*&&[^*subtract*]]›.

Пересечение и разность символьных классов удобно использовать, в частности, при работе со свойствами Юникода, блоками и алфавитами. Например, выражению ‹\p{IsThai}› соответствует любой символ тайского алфавита. Выражению ‹\p{N}› соответствует любой символ, у которого присутствует свойство Number. Последовательности ‹[\p{InThai}&&[\p{N}]]› соответствует любая из 10 цифр тайского алфавита.

Если вас интересуют различия между возможными комбинациями «\p» в регулярных выражениях, их описание вы найдете в рецепте 2.7, где также подробно описываются тонкости работы со свойствами Юникода.

См. также

Рецепт 2.2, где описывается, как организовать поиск непечатаемых символов. В рецепте 2.7 рассказывается об особенностях сопоставления с символами Юникода. Синтаксис определения непечатаемых символов и символов Юникода с успехом можно использовать в символьных классах.

В разделе «Bat, cat или rat» в рецепте 5.3 описываются некоторые наиболее частые ошибки в символьных классах, допускаемые теми, кто только начинает осваивать регулярные выражения.

2.4. Сопоставление с любым символом

Этот рецепт объясняет достоинства и недостатки метасимвола «точка».

Задача

Создать регулярное выражение, которому соответствует любой символ, заключенный в одиночные кавычки. Одному решению должен соответствовать любой одиночный символ, за исключением конца строки, заключенный в одиночные кавычки. Второму решению должен соответствовать действительно любой символ, включая символы конца строки.

Решение

Любой символ, за исключением символа конца строки

Параметры: нет (параметр «точке соответствуют границы строк» должен быть сброшен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Любой символ, включая символы конца строки

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
'[\s\S]'
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Любой символ, за исключением символа конца строки

Точка – это один из старейших и простейших элементов регулярных выражений. Она означает обязательное соответствие любому одиночному символу.

Однако не всегда очевидно, что означает фраза *любой символ*. Самые ранние инструменты, обеспечивающие поддержку регулярных выражений, обрабатывали содержимое файлов строку за строкой, поэтому отсутствовала потребность искать соответствие с концом строки в испытуемом тексте. Языки программирования, рассматриваемые в этой книге, обрабатывают текст целиком, как единое целое, независимо от наличия в нем разрывов строк. Если действительно возникает потребность производить обработку текста построчно, можно дополнить программу кодом, который будет разбивать исходный текст на массив строк и затем применять регулярное выражение к каждой строке в массиве. Как это делается, показано в рецепте 3.21 в следующей главе.

Ларри Уолл (Larry Wall), создатель языка Perl, сохранил в языке Perl традиционное поведение инструментов построчной обработки текста, где точка никогда не соответствовала концу строки. Все остальные диалекты, рассматриваемые в этой книге, последовали его примеру. То есть регулярному выражению ‹.› соответствует любой одиночный символ, *за исключением* символа перевода строки.

Любой символ, включая символы конца строки

Если необходимо позволить регулярному выражению обрабатывать сразу несколько строк, следует включить параметр «точке соответствуют границы строк». Этот параметр маскируется под разными названиями. В языке Perl и во многих других он носит сбивающее с толку название «режим единственной строки» (single line mode), тогда как в Java он называется «режим ‘точка – это все’» (‘dot all’ mode). Подробное описание этого вопроса вы найдете в рецепте 3.4 в следующей главе. Независимо от того, как называется этот параметр в вашем любимом языке программирования, называйте его про себя режимом «точке соответствуют границы строк». Это все, что дает данный параметр.

Для JavaScript, где отсутствует параметр «точке соответствуют границы строк», необходимо альтернативное решение. Как описывается в рецепте 2.3, метасимволу ‹\s› соответствуют любые пробельные символы, тогда как метасимволу ‹\S› соответствуют любые другие символы, не соответствующие метасимволу ‹\s›. Объединив их, можно получить символьный класс ‹[\s\S]›, которому будут соответствовать любые символы, включая символы конца строки. Тот же эффект дают символьные классы ‹[\d\D]› и ‹[\w\W]›.

Злоупотребление точкой

Точка является элементом регулярных выражений, которым злоупотребляют наиболее часто. Выражение ‹\d\d.\d\d.\d\d› представляет не лучший способ поиска дат. Ему прекрасно соответствует дата 05/16/08, но ему также соответствует текст 99/99/99. Хуже того, ему соответствует число 12345678.

Регулярное выражение, которому соответствуют только корректные даты, – это тема для более поздней главы (см. рецепт 4.5). Но заменить точку на более подходящий символьный класс совсем несложно. Выражение ‹\d\d[/.\-]\d\d[/.\-]\d\d› позволяет использовать в качестве символа-разделителя слэш, точку или дефис. Данному регулярному выражению по-прежнему соответствует текст 99/99/99, но число 12345678 уже не соответствует.

Это просто совпадение, что в предыдущем примере точка включена в символьные классы. Внутри символьных классов точка интерпретируется буквально. В этом есть определенный смысл, так как в некоторых странах, например в Германии, точка используется в качестве символа-разделителя в датах.

Точку следует использовать, только если действительно допускается совпадение с любым символом. Во всех остальных случаях лучше использовать символьные классы и инвертированные символьные классы.

Варианты

Ниже показано, как обеспечить совпадение с символом в одиночных кавычках, включая границы строк, с помощью встроенного модификатора режима:

`(?s)'.'`

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python

`(?m)'.'`

Параметры: нет
Диалект: Ruby

Если не удалось включить режим «точке соответствуют границы строк» вне регулярного выражения, можно добавить модификатор режима в начало выражения. В рецепте 2.1 в подразделе «Поиск без учета регистра символов» мы объяснили концепцию модификаторов режима и говорили, что в JavaScript они не поддерживаются.

‹(?s)› – это модификатор режима «точке соответствуют границы строк» в .NET, Java, XRegExp, PCRE, Perl и Python. Символ s здесь обозначает режим «single line» (единственная строка), под которым в языке Perl подразумевается режим «точке соответствуют границы строк».

Терминология настолько запутывающая, что разработчик механизма регулярных выражений в языке Ruby допустил ошибку. Для включения режима «точке соответствуют границы строк» в Ruby используется модификатор `(?m)`. Здесь используется другой символ, но функциональность та же самая. Новый механизм в Ruby 1.9 продолжает использовать модификатор `(?m)` для включения режима «точке соответствуют границы строк». Значение модификатора `(?m)` в языке Perl совершенно иное; оно описывается в рецепте 2.5.

См. также

Во многих случаях в действительности требуется не совпадение с *любым* символом, а с любым символом из некоторого множества. Как это реализовать, описывается в рецепте 2.3.

Рецепт 3.4, где объясняется, как устанавливать параметры, такие как «точке соответствуют границы строк» в исходном программном коде.

При работе с текстом, содержащим символы Юникода, может оказаться предпочтительнее использовать метасимвол `\X`, соответствующий графеме Юникода, а не «точку», которая совпадает с кодовым пунктом Юникода. Подробнее об этом рассказывается в рецепте 2.7.

2.5. Сопоставление в начале и/или в конце строки

Задача

Создать четыре регулярных выражения. Первому выражению должно соответствовать слово `alpha`, но только если оно находится в самом начале испытуемого текста. Второму выражению должно соответствовать слово `omega`, но только если оно находится в самом конце испытуемого текста. Третьему выражению должно соответствовать слово `begin`, но только если оно находится в начале строки. Четвертому выражению должно соответствовать слово `end`, но только если оно находится в конце строки.

Решение

В начале испытуемого текста

```
^alpha
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\Aalpha
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В конце испытуемого текста

`omega$`

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

`omega\Z`

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В начале строки

`^begin`

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В конце строки

`end$`

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Якорные метасимволы и строки

Метасимволы регулярных выражений ‹^›, ‹$›, ‹\A›, ‹\Z› и ‹\z› называются *якорями*. Они не соответствуют каким-либо символам; вместо этого они обозначают определенные позиции, фактически осуществляя привязку регулярного выражения к этим позициям.

Строка – это часть испытуемого текста, которая находится между начальной и конечной границами строки. Если в испытуемом тексте отсутствуют границы строк, то весь текст рассматривается как одна строка. Так, следующий текст состоит из четырех строк, по одной для слов `one` и `two`, затем следует пустая строка и далее строка со словом `four`:

```
one
two

four
```

В программном коде этот текст можно было бы представить так: `one`LF `two`LF LF `four`.

В начале испытуемого текста

Якорному метасимволу ‹\A› всегда соответствует точка самого начала испытуемого текста, непосредственно перед первым символом. Это единственное место в тексте, соответствующее данному метасимволу. Поместив

‹\A› в начало регулярного выражения, можно проверить, начинается ли испытуемый текст с искомого текста. Символ «A» должен быть символом верхнего регистра.

Диалект JavaScript не поддерживает метасимвол ‹\A›.

Якорный метасимвол ‹^› является эквивалентом метасимвола ‹\A› при условии, что параметр «символам ^ и $ соответствуют границы строк» отключен. Этот параметр отключен по умолчанию во всех диалектах регулярных выражений, за исключением Ruby. Ruby не предоставляет возможность отключать этот параметр.

Если вы не используете диалект JavaScript, мы рекомендуем всегда вместо метасимвола ‹^› использовать метасимвол ‹\A›. Значение метасимвола ‹\A› никогда не изменяется, что позволяет избегать путаницы и ошибок при установке параметров регулярных выражений.

В конце испытуемого текста

Якорным метасимволам ‹\Z› и ‹\z› всегда соответствует позиция в конце испытуемого текста непосредственно за последним символом. Поместив ‹\Z› или ‹\z› в конец регулярного выражения, можно проверить, оканчивается ли испытуемый текст искомым текстом.

Диалекты .NET, Java, PCRE, Perl и Ruby поддерживают оба метасимвола, ‹\Z› и ‹\z›. Диалект Python поддерживает только метасимвол ‹\Z›. Диалект JavaScript не поддерживает ни один из них.

Различия между метасимволами ‹\Z› и ‹\z› начинают проявляться, когда в испытуемом тексте последним символом является символ конца строки. В этом случае метасимволу ‹\Z› соответствует самый конец испытуемого текста, после завершающего символа конца строки, а также позиция непосредственно перед этим символом конца строки. Преимущество состоит в том, что можно выполнить поиск ‹omega\Z›, не беспокоясь об удалении завершающего символа конца строки в конце испытуемого текста. При чтении содержимого файла строка за строкой одни инструменты включают символ конца строки, другие – нет; метасимвол ‹\Z› позволяет ликвидировать это различие. Метасимволу ‹\z› соответствует только самый конец испытуемого текста, поэтому он не будет совпадать с текстом в регулярном выражении, если в нем имеется завершающий символ конца строки.

Якорный метасимвол ‹$› является эквивалентом метасимвола ‹\Z› при условии, что параметр «символам ^ и $ соответствуют границы строк» отключен. Этот параметр отключен по умолчанию во всех диалектах регулярных выражений, за исключением Ruby. Ruby не предоставляет возможность отключать этот параметр. Так же как и ‹\Z›, метасимволу ‹$› соответствует самый конец испытуемого текста, а также позиция перед завершающим символом конца строки.

Чтобы прояснить все эти тонкости и неясности, рассмотрим пример на языке Perl. Предположим, что переменная $/ (текущий символ-разде-

литель записей) имеет значение по умолчанию \n, тогда следующая инструкция на языке Perl прочитает одну строку, введенную с терминала (из потока стандартного ввода):

```
$line = <>;
```

В языке Perl символ перевода строки будет сохранен в переменной $line. Вследствие этого регулярное выражение, такое как ‹end•of•input.\z›, не совпадет с содержимым переменной. Но совпадения будут обнаружены выражениями ‹end•of•input.\Z› и ‹end•of•input.$›, потому что они игнорируют завершающий символ перевода строки.

Чтобы упростить обработку данных, программисты на языке Perl часто принудительно отбрасывают символ перевода строки инструкцией:

```
chomp $line;
```

После выполнения этой операции совпадение будет обнаружено всеми тремя выражениями. (Технически инструкция chomp удаляет из строки текущий символ-разделитель записей.)

Если вы не пользуетесь JavaScript, мы рекомендуем всегда применять метасимвол ‹\Z› вместо ‹$›. Значение метасимвола ‹\Z› никогда не изменяется, что позволяет избегать путаницы и ошибок при установке параметров регулярных выражений.

В начале строки

По умолчанию метасимволу ‹^› соответствует позиция в начале испытуемого текста, как и метасимволу ‹\A›. Только в Ruby метасимволу ‹^› всегда соответствует начало строки. Во всех остальных диалектах необходимо включать дополнительный параметр, чтобы символ крышки и знак доллара совпадали с границами строк. Этот параметр обычно называется «многострочным» (multiline) режимом.

Не следует путать этот режим с режимом «единственной строки», который более известен как режим «точке соответствуют границы строк». «Многострочный» режим оказывает влияние только на поведение символа крышки и знака доллара, а режим «единственной строки» оказывает влияние только на поведение точки, как разъяснялось в рецепте 2.4. Нет ничего невозможного в том, чтобы одновременно включить «многострочный» режим и режим «единственной строки». По умолчанию оба параметра отключены.

При корректной установке параметра метасимволу ‹^› будет соответствовать позиция в начале каждой строки испытуемого текста. Строго говоря, метасимвол будет совпадать с позицией непосредственно перед самым первым символом в файле, что происходит всегда, а также с позициями после каждого символа конца строки. Символ крышки в выражении ‹\n^› избыточен, потому что ‹^› всегда совпадает с позицией после ‹\n›.

В конце строки

По умолчанию метасимволу ‹$› соответствует позиция только в конце испытуемого текста или перед завершающим символом конца строки, как и метасимволу ‹\Z›. Только в Ruby метасимволу ‹$› всегда соответствует конец каждой строки. Во всех остальных диалектах необходимо включать дополнительный «многострочный» режим, чтобы символ крышки и знак доллара совпадали с границами строк.

При корректной установке параметра метасимволу ‹$› будет соответствовать позиция в конце каждой строки испытуемого текста. (Безусловно, он также будет совпадать с позицией за самым последним символом в тексте, потому что эта позиция всегда считается концом строки.) Символ доллара в выражении ‹$\n› избыточен, потому что ‹$› всегда совпадает с позицией перед ‹\n›.

Совпадения нулевой длины

Нет ничего необычного в том, чтобы регулярное выражение содержало только один или более якорных метасимволов. Такое регулярное выражение будет находить совпадение нулевой длины в каждой позиции, где будет обнаруживаться соответствие якорному метасимволу. При объединении нескольких якорных метасимволов все они должны совпадать с одной и той же позицией, чтобы регулярное выражение находило соответствие.

Такое регулярное выражение может использоваться в операции поиска с заменой. Замещение метасимволов ‹\A› или ‹\Z› позволяет добавлять что-либо в начало или в конец испытуемого текста. Замещение метасимволов ‹^› или ‹$› в режиме «символам ^ и $ соответствуют границы строк» позволяет добавлять что-либо в начало или в конец каждой строки испытуемого текста.

Комбинация двух якорных метасимволов позволит проверить наличие пустых строк или отсутствие ввода. Комбинации ‹\A\Z› соответствует пустая строка, а также строка, состоящая из единственного символа перевода строки. Комбинации ‹\A\z› соответствует только пустая строка. Комбинации ‹^$› в режиме «символам ^ и $ соответствуют границы строк» будет соответствовать каждая пустая строка в испытуемом тексте.

Варианты

`(?m)^begin`

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python

`(?m)end$`

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python

Если нет возможности включить режим «символам ^ и $ соответствуют границы строк» вне регулярного выражения, в начало регулярного выражения можно поместить модификатор режима. О модификаторах режима и об отсутствии их поддержки в JavaScript рассказывалось в рецепте 2.1 в подразделе «Поиск без учета регистра символов».

‹(?m)› – это модификатор режима «символам ^ и $ соответствуют границы строк» в диалектах .NET, Java, XRegExp, PCRE, Perl и Python. Символ m соответствует названию «multiline» (многострочный) режим, под которым в языке Perl подразумевается режим «символам ^ и $ соответствуют границы строк».

Как уже говорилось выше, терминология оказалась настолько запутывающей, что разработчик механизма регулярных выражений в языке Ruby допустил ошибку. Модификатор ‹(?m)› в языке Ruby используется для включения режима «точке соответствуют границы строк». То есть в языке Ruby модификатор ‹(?m)› не оказывает влияния на поведение якорных метасимволов ‹^› и ‹$›. В Ruby метасимволам ‹^› и ‹$› всегда соответствуют начало и конец каждой строки.

За исключением путаницы в символах модификаторов, в языке Ruby выбор применения метасимволов ‹^› и ‹$› исключительно к границам строк следует признать удачным. Если вы не используете JavaScript, мы рекомендуем придерживаться такого же выбора в своих регулярных выражениях.

Ян Гойвертс следовал этой идее при разработке программ EditPad Pro и PowerGREP. В них вы не найдете флажок с надписью ^ and $ match at line breaks (символам ^ и $ соответствуют границы строк), хотя имеется флажок с надписью dot matches line breaks (точке соответствуют границы строк). Если добавить в начало регулярного выражения модификатор ‹(?-m)›, то, чтобы привязать регулярное выражение к началу или к концу файла, необходимо будет использовать метасимволы ‹\A› и ‹\Z›.

См. также

Рецепт 3.4, где объясняется, как устанавливать такие параметры, как «символам ^ и $ соответствуют границы строк», в исходном коде программ.

Рецепт 3.21, где показано, как использовать процедурный код для построчной обработки текста с помощью регулярного выражения.

2.6. Сопоставление с целыми словами

Задача

Создать регулярное выражение, которому соответствовало бы слово cat в тексте `My cat is brown`, но которое не находило бы соответствие в словах `category` или `bobcat`. Создать еще одно регулярное выражение, которому

соответствовала бы последовательность cat в тексте staccato, но которое не находило бы соответствие ни в одном из трех предыдущих случаев.

Решение

На границах слова

```
\bcat\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Не на границах слова

```
\Bcat\B
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

На границах слова

Метасимвол ‹\b› называется *границей слова*. Ему соответствует позиция начала или конца слова. Сам по себе этот метасимвол дает совпадение нулевой длины. Метасимвол ‹\b› является *якорным*, точно так же как и метасимволы, представленные в предыдущем разделе.

Строго говоря, метасимволу ‹\b› соответствуют следующие три позиции:

- Перед первым символом испытуемого текста, если первый символ является символом слова.
- После последнего символа испытуемого текста, если последний символ является символом слова.
- Между двумя символами испытуемого текста, когда один из них является символом слова, а другой – нет.

Чтобы выполнить поиск «только целого слова» с помощью регулярного выражения, нужно просто поместить искомое слово между двумя границами слов, как это предложено в решении ‹\bcat\b›. Первый метасимвол ‹\b› требует, чтобы символ ‹c› находился в самом начале строки или после символа, не являющегося символом слова. Второй метасимвол ‹\b› требует, чтобы символ ‹t› находился в самом конце строки или перед символом, не являющимся символом слова.

Символы конца строки не являются символами слова. Поэтому метасимвол ‹\b› будет совпадать с позицией после символа конца строки, если сразу же вслед за ним располагается символ слова. Он также будет совпадать с позицией перед символом конца строки, если ему непосредственно предшествует символ слова. Благодаря этому слово, занимающее всю строку, будет обнаружено в процессе поиска «только целых слов». На поведение метасимвола ‹\b› не влияет выбор «многострочного»

режима или наличие модификатора ‹(?m)›, что является одной из причин, почему в этой книге «многострочный» режим сопоставляется с режимом «символам ^ и $ соответствуют границы строк».

Ни в одном из диалектов, обсуждаемых в этой книге, нет отдельных метасимволов, которым соответствовала бы позиция только в начале или только в конце слова. Однако в них нет никакой необходимости, если только вам не потребуется регулярное выражение, не содержащее ничего, кроме границы слова. Символы перед или после метасимвола ‹\b› в регулярном выражении помогут указать, с какой позицией будет совпадать ‹\b›. Метасимвол ‹\b› в выражениях ‹\bx› и ‹!\b› мог бы совпадать только с началом слова. Метасимвол ‹\b› в выражениях ‹x\b› и ‹\b!› мог бы совпадать только с концом слова. А выражения ‹x\bx› и ‹!\b!› никогда не будут находить соответствий.

Если действительно потребуется обеспечить совпадение только в начале слова или только в конце слова, для этого можно использовать опережающую или ретроспективную проверку. Подробнее об этих проверках рассказывается в рецепте 2.16. Этот прием невозможен в диалектах JavaScript и Ruby 1.8, так как они не поддерживают ретроспективные проверки. Регулярное выражение ‹(?<!\w)(?=\w)› соответствует началу слова – оно требует, чтобы символ перед позицией совпадения не был символом слова, а символ за позицией совпадения, напротив, был символом слова. Выражение ‹(?<=\w)(?!\w)› действует с точностью до наоборот: оно соответствует концу слова, требуя, чтобы предыдущий символ был символом слова, а следующий, напротив, не был символом слова. Важно заметить, что для проверки отсутствия символа слова должна использоваться отрицательная проверка с помощью метасимвола ‹\w›, а не положительная – с помощью метасимвола ‹\W›. Выражение ‹(?<!\w)› соответствует началу сроки, так как перед началом строки отсутствует символ слова (как и любой другой символ). Но выражение ‹(?<=\W)› никогда не совпадет с началом строки. Выражение ‹(?!\w)› соответствует концу строки по той же самой причине. Поэтому две наши конструкции на основе проверок будут корректно совпадать с началом строки, если она начинается словом, и с концом строки, если она завершается словом.

Не на границах слова

Метасимволу ‹\B› соответствует любая позиция в испытуемом тексте, которая не соответствует метасимволу ‹\b›. Отсюда следует, что метасимволу ‹\B› соответствует любая позиция, не являющаяся началом или концом слова.

Строго говоря, метасимволу ‹\B› соответствуют следующие пять позиций:

- Перед первым символом испытуемого текста, если первый символ не является символом слова.
- После последнего символа испытуемого текста, если последний символ не является символом слова.

- Между двумя символами слова.
- Между двумя символами, не являющимися символами слова.
- Пустая строка.

Выражению ‹\Bcat\B› соответствует последовательность символов cat в тексте staccato, но не в текстах My cat is brown, category или bobcat.

Чтобы выполнить поиск с условием, противоположным требованию «только целое слово» (то есть чтобы совпадение обнаруживалось в словах staccato, category и bobcat, но не обнаруживалось в тексте My cat is brown), необходимо с помощью операции выбора объединить подвыражения ‹\Bcat› и ‹cat\B› в выражение ‹\Bcat|cat\B›. Подвыражению ‹\Bcat› соответствует последовательность символов cat в словах staccato и bobcat. Подвыражению ‹cat\B› соответствует последовательность символов cat в слове category (и в слове staccato, если подвыражение ‹\Bcat› не позаботилось о нем раньше). Оператор выбора рассматривается в рецепте 2.8.

Символы слов

За всеми этими разговорами о границах слов мы ничего не сказали о том, что такое *символ слова*. Символ слова – это символ, который может являться частью слова. В рецепте 2.3 в подразделе «Сокращения» говорилось о том, какие символы включены в состав символьного класса ‹\w›, который соответствует одному символу слова. К сожалению, для метасимвола ‹\b› история выглядит иначе.

Несмотря на то что все диалекты, рассматриваемые в этой книге, поддерживают метасимволы ‹\b› и ‹\B›, тем не менее они по-разному определяют, какие символы являются символами слова.

В диалектах .NET, JavaScript, PCRE, Perl, Python и Ruby метасимволу ‹\b› соответствует позиция между двумя символами, один из которых соответствует символьному классу ‹\w›, а другой – символьному классу ‹\W›. Метасимволу ‹\B› всегда соответствует позиция между двумя символами, когда оба они соответствуют либо символьному классу ‹\w›, либо символьному классу ‹\W›.

В диалектах JavaScript, PCRE и Ruby символами слов считаются только символы ASCII. Символьный класс ‹\w› в них идентичен классу ‹[a-zA-Z0-9_]›. При использовании этих диалектов можно выполнить поиск «только целого слова» в тексте, написанном на языке, где используются исключительно символы от A до Z без диакритических знаков, например на английском. Но эти диалекты не могут использоваться для поиска «только целых слов» в текстах на других языках, таких как испанский или русский.

Диалект .NET интерпретирует буквы и цифры из любых алфавитов как символы слова. В этих диалектах можно выполнить поиск «только целого слова» в текстах на любом языке, включая и те, которые не используют латинский алфавит.

В языке Python имеется возможность выбора. В Python 2.x символы, не входящие в набор ASCII, могут включаться в поиск, только если при создании регулярного выражения был передан флаг `UNICODE` или `U`. В Python 3.x символы, не входящие в набор ASCII, включаются по умолчанию, но их можно исключить, установив флаг `ASCII` или `A`. Эти флаги в равной степени воздействует на метасимволы `‹\b›` и `‹\w›`.

В языке Perl включение в символьный класс `‹\w›` символов ASCII или Юникода, цифр и подчеркиваний зависит от версии Perl и флагов `/adlu`. Подробнее об этом рассказывается в рецепте 2.3, в подразделе «Сокращения». Во всех версиях Perl поведение метасимвола `‹\b›` полностью соответствует символьному классу `‹\w›`.

Диалект Java отличается некоторой непоследовательностью. В версиях Java 4–6 классу `‹\w›` соответствуют только символы ASCII. В Java 7 по умолчанию классу `‹\w›` соответствуют только символы ASCII, но если установить флаг `UNICODE_CHARACTER_CLASS`, ему будут соответствовать и символы Юникода. Однако метасимвол `‹\b›` поддерживает Юникод во всех версиях Java и может применяться при работе с любыми алфавитами. В версиях Java 4–6 выражению `‹\b\w\b›` соответствует одна буква латинского алфавита, цифра или символ подчеркивания, который не является частью слова ни на одном языке. Выражение `‹\bкошка\b›` корректно совпадет со словом `кошка` в тексте на русском языке, потому что метасимвол `‹\b›` поддерживает Юникод. Но выражение `‹\w+›` не совпадет ни с одним русским словом в Java 4–6, потому что класс `‹\w›` поддерживает только символы ASCII.

См. также

Рецепт 2.3, где описывается, какие символы соответствуют сокращенному символьному классу `‹\w›`, соответствующему символу слова.

Рецепт 5.1, где демонстрируется, как использовать границы слов для поиска совпадений с целыми словами и как преодолевать проблемы, связанные с различиями поведения границ слов в разных диалектах регулярных выражений.

2.7. Кодовые пункты Юникода, категории, блоки и алфавиты

Задача

С помощью регулярного выражения отыскать символ торговой марки (™), указав в нем кодовый пункт Юникода вместо копирования и вставки фактического символа. При желании можно копировать и вставлять сам символ; в конце концов, символ торговой марки – это всего лишь обычный литерал, несмотря на то, что его нельзя ввести с клавиатуры непосредственно. Литералы обсуждались в рецепте 2.1.

Создать регулярное выражение, которому соответствовали бы любые символы из категории Юникода «Currency Symbol» (символ денежной единицы).

Создать регулярное выражение, которому соответствовали бы любые символы, принадлежащие блоку Юникода «Greek Extended».

Создать регулярное выражение, которому соответствовали бы любые символы, которые в соответствии со стандартом Юникода принадлежат греческому алфавиту.

Создать регулярное выражение, которому соответствовали бы графемы, то есть символы, состоящие из базового символа и дополнительных комбинационных знаков.

Решение

Кодовый пункт Юникода

```
\u2122
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python, Ruby 1.9

```
\U00002122
```

Параметры: нет
Диалекты: Python

Данные регулярные выражения будет работать в Python 2.x, только если они будут оформлены как строка Юникода: `u"\u2122"` или `u"\U00002122"`.

```
\x{2122}
```

Параметры: нет
Диалекты: Java 7, PCRE, Perl

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8; в языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона `/u`.

```
\u{2122}
```

Параметры: нет
Диалекты: Ruby 1.9

В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Категория Юникода

```
\p{Sc}
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Ruby 1.9

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, в языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона /u. Диалекты JavaScript и Python не поддерживают категории Юникода. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Блок Юникода

`\p{IsGreekExtended}`

Параметры: нет
Диалекты: .NET, Perl

`\p{InGreekExtended}`

Параметры: нет
Диалекты: Java, XRegExp, Perl

Диалекты JavaScript, PCRE, Python и Ruby 1.9 не поддерживают блоки Юникода. Они поддерживают кодовые пункты Юникода, которые можно использовать для сопоставления с блоками, как показано в разделе «Варианты» в этом рецепте. Диалект XRegExp добавляет поддержку блоков Юникода в JavaScript.

Алфавит в Юникоде

`\p{Greek}`

Параметры: нет
Диалекты: XRegExp, PCRE, Perl, Ruby 1.9

`\p{IsGreek}`

Параметры: нет
Диалекты: Java 7, Perl

Для работы с алфавитами необходима библиотека PCRE версии 6.5 или выше, а кроме того, библиотека должна быть скомпилирована с поддержкой UTF-8. В языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона /u. Диалекты .NET, JavaScript и Python не поддерживают алфавиты Юникода. Диалект XRegExp добавляет поддержку алфавитов Юникода в JavaScript. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Графема в Юникоде

`\X`

Параметры: нет
Диалекты: PCRE, Perl

В диалектах PCRE и Perl имеется специальный метасимвол, соответствующий графемам. Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8. В языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона /u.

`(?>\P{M}\p{M}*)`

Параметры: нет
Диалекты: .NET, Java, Ruby 1.9

`(?:\P{M}\p{M}*)`

Параметры: нет
Диалекты: XRegExp

Диалекты .NET, Java, XRegExp и Ruby 1.9 не имеют метасимвола, соответствующего графемам. Но они поддерживают категории Юникода, посредством которых можно имитировать сопоставление с графемами.

Диалекты JavaScript (без XRegExp) и Python не поддерживают графемы Юникода. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Обсуждение

Кодовый пункт Юникода

Кодовый пункт – это одна запись в базе данных символов Юникода. Кодовый пункт – это не *символ*; впрочем, это зависит от того, какой смысл вкладывается в слово «символ». То, что на экране выглядит как символ, в Юникоде называется *графемой*.

Кодовый пункт Юникода U+2122 представляет символ «знака торговой марки». Его можно отыскать с помощью конструкции ‹`\u2122`›, ‹`\u{2122}`› или ‹`\x{2122}`› в зависимости от используемого диалекта регулярных выражений.

При использовании метасимвола ‹`\u`› необходимо указывать точно четыре шестнадцатеричные цифры. Это означает, что его можно использовать для поиска кодовых пунктов Юникода в диапазоне от U+0000 до U+FFFF.

В конструкции ‹`\u{…}`› и ‹`\x{…}`› допускается указывать в фигурных скобках от одной до шести шестнадцатеричных цифр, что обеспечивает поддержку всех кодовых пунктов в диапазоне от U+000000 до U+10FFFF. Кодовый пункт U+00E0 будет соответствовать как выражению ‹`\x{E0}`›, так и выражению ‹`\x{00E0}`›. Кодовые пункты U+100000 и выше используются очень редко и в настоящее время весьма слабо поддерживаются шрифтами и операционными системами.

Механизм регулярных выражений в Python не поддерживает кодовые пункты Юникода. Литералы строк Юникода в Python 2.x и литералы текстовых строк в Python 3.x позволяют вставлять в них экранированные последовательности, соответствующие кодовым пунктам Юникода. Последовательности от `\u0000` до `\uFFFF` представляют кодовые пункты Юникода от U+0000 до U+FFFF. Последовательности от `\U00000000` до `\U0010FFFF` представляют все кодовые пункты Юникода. После метасимвола `\U` необходимо указывать точно восемь шестнадцатеричных цифр, даже для кодовых пунктов Юникода ниже U+10FFFF.

При составлении регулярных выражений в виде строковых литералов в программе на языке Python допускается использовать экранированные последовательности ‹\u2122› и ‹\U00002122› непосредственно в регулярных выражениях. Если регулярные выражения извлекаются из файла или вводятся пользователем, эти экранированные последовательности, обозначающие кодовые пункты Юникода, не будут работать, если передать прочитанную из файла или принятую от пользователя строку непосредственно в вызов `re.compile()`. В Python 2.x экранированные последовательности Юникода можно декодировать вызовом `string.decode('unicode-escape')`. В Python 3.x можно вызвать `string.encode('utf-8').decode('unicode-escape')`.

Кодовые пункты могут использоваться внутри символьных классов и за их пределами.

Категории Юникода

Каждый кодовый пункт Юникода принадлежит единственной *категории Юникода*. Всего существует 30 категорий Юникода, определяемых одно- или двубуквенными идентификаторами. Эти категории сгруппированы в 7 суперкатегорий, определяемых однобуквенными идентификаторами:

‹\p{L}› Любая буква любого языка.

‹\p{Ll}› Любая строчная буква, для которой имеется прописной вариант.

‹\p{Lu}› Прописная буква, для которой имеется строчный вариант.

‹\p{Lt}› Начальная буква слова при условии, что капитализируется только первая буква слова.

‹\p{Lm}› Специальный символ, который используется как буква.

‹\p{Lo}› Символ или идеограмма, для которого отсутствуют варианты нижнего и верхнего регистра.

‹\p{M}› Символ, который должен объединяться с другим символом (диакритические знаки, умляуты, описывающие рамки и пр.).

‹\p{Mn}› Символ, который должен объединяться с другим символом, не требующий дополнительного пространства (например, диакритические знаки, умляуты и пр.).

‹\p{Mc}› Символ, который должен объединяться с другим символом, требующий дополнительного пространства (например, огласовки во многих восточных алфавитах).

‹\p{Me}› Символ, описанный вокруг другого символа (окружность, квадрат, клавишный колпачок и пр.).

‹\p{Z}› Любые типы пробельных символов или невидимых символов-разделителей.

‹\p{Zs}› Невидимые пробельные символы, занимающие дополнительное пространство.

`\p{Zl}` Символ-разделитель строк U+2028.

`\p{Zp}` Символ-разделитель абзацев U+2029.

`\p{S}` Математические символы, знаки денежных единиц, декоративные элементы, символы для рисования рамок и пр.

`\p{Sm}` Любые математические символы.

`\p{Sc}` Любые знаки денежных единиц.

`\p{Sk}` Комбинационный знак (метка) как самостоятельный символ.

`\p{So}` Различные символы, которые не являются математическими знаками, знаками денежных единиц или комбинационными символами.

`\p{N}` Различные числовые символы в любом алфавите.

`\p{Nd}` Цифры от 0 до 9 в любом алфавите, за исключением идеографических алфавитов.

`\p{Nl}` Числа, которые выглядят как буквы, например римские цифры.

`\p{No}` Надстрочные или подстрочные цифры или числа, не являющиеся цифрами 0...9 (кроме чисел из идеографических алфавитов).

`\p{P}` Различные знаки пунктуации.

`\p{Pd}` Различные тире и дефисы.

`\p{Ps}` Различные открывающие скобки.

`\p{Pe}` Различные закрывающие скобки.

`\p{Pi}` Различные открывающие кавычки.

`\p{Pf}` Различные закрывающие кавычки.

`\p{Pc}` Знаки пунктуации, например подчеркивание, соединяющее слова.

`\p{Po}` Различные знаки пунктуации, не являющиеся дефисами, тире, скобками, кавычками или символами, соединяющими слова.

`\p{C}` Неотображаемые управляющие символы и неиспользуемые кодовые пункты.

`\p{Cc}` Управляющие символы ASCII или Latin-1 в диапазонах 0x00...0x1F и 0x7F...0x9F.

`\p{Cf}` Неотображаемые символы, являющиеся признаками форматирования.

`\p{Co}` Кодовые пункты, предназначенные для закрытого применения.

`\p{Cs}` Одна половина суррогатной пары в кодировке UTF-16.

`\p{Cn}` Кодовые пункты, которым не присвоены символы.

Комбинация `\p{Ll}` совпадает с одиночным кодовым пунктом из категории `Ll`, или «lowercase letter» (символ нижнего регистра). Комбина-

ция ‹\p{L}› является сокращенной формой записи символьного класса ‹[\p{Ll}\p{Lu}\p{Lt}\p{Lm}\p{Lo}]›, которому соответствует любой кодовый пункт из категории «letter» (буква).

Метасимвол ‹\P› – это инвертированная версия метасимвола ‹\p›. Комбинации ‹\P{Ll}› соответствует одиночный кодовый пункт, не относящийся к категории Ll. Комбинации ‹\P{L}› соответствует единственный кодовый пункт, не относящийся к суперкатегории «letter» (буква). Эта комбинация не является аналогом символьного класса ‹[\P{Ll}\P{Lu}\P{Lt}\P{Lm}\P{Lo}]›, которому соответствует любой кодовый пункт. Комбинации ‹\P{Ll}› соответствуют кодовые пункты из категории Lu (или любым другим категориям, за исключением Ll), тогда как комбинация ‹\P{Lu}› будет совпадать и с кодовыми пунктами из категории Ll. Объединение только этих двух комбинаций в символьный класс уже обеспечивает совпадение с любыми возможными кодовыми пунктами.

В Perl, а также в библиотеке PCRE версии 6.5 и выше комбинацию ‹\p{L&}› можно использовать как сокращение для выражения ‹[\p{Ll}\p{Lu}\p{Lt}]›, которому соответствуют все буквы из всех алфавитов, имеющие варианты верхнего и нижнего регистров.

Блок Юникода

Все кодовые пункты в базе данных символов Юникода подразделяются на блоки. Каждый содержит отдельный диапазон кодовых пунктов. Кодовые пункты с U+0000 по U+FFFF делятся на 156 кодовых блоков в версии 6.1 стандарта Юникода:

```
‹U+0000...U+007F \p{InBasicLatin}›
‹U+0080...U+00FF \p{InLatin-1Supplement}›
‹U+0100...U+017F \p{InLatinExtended-A}›
‹U+0180...U+024F \p{InLatinExtended-B}›
‹U+0250...U+02AF \p{InIPAExtensions}›
‹U+02B0...U+02FF \p{InSpacingModifierLetters}›
‹U+0300...U+036F \p{InCombiningDiacriticalMarks}›
‹U+0370...U+03FF \p{InGreekandCoptic}›
‹U+0400...U+04FF \p{InCyrillic}›
‹U+0500...U+052F \p{InCyrillicSupplement}›
‹U+0530...U+058F \p{InArmenian}›
‹U+0590...U+05FF \p{InHebrew}›
‹U+0600...U+06FF \p{InArabic}›
‹U+0700...U+074F \p{InSyriac}›
‹U+0750...U+077F \p{InArabicSupplement}›
‹U+0780...U+07BF \p{InThaana}›
‹U+07C0...U+07FF \p{InNKo}›
‹U+0800...U+083F \p{InSamaritan}›
‹U+0840...U+085F \p{InMandaic}›
‹U+08A0...U+08FF \p{InArabicExtended-A}›
‹U+0900...U+097F \p{InDevanagari}›
‹U+0980...U+09FF \p{InBengali}›
```

```
‹U+0A00...U+0A7F \p{InGurmukhi}›
‹U+0A80...U+0AFF \p{InGujarati}›
‹U+0B00...U+0B7F \p{InOriya}›
‹U+0B80...U+0BFF \p{InTamil}›
‹U+0C00...U+0C7F \p{InTelugu}›
‹U+0C80...U+0CFF \p{InKannada}›
‹U+0D00...U+0D7F \p{InMalayalam}›
‹U+0D80...U+0DFF \p{InSinhala}›
‹U+0E00...U+0E7F \p{InThai}›
‹U+0E80...U+0EFF \p{InLao}›
‹U+0F00...U+0FFF \p{InTibetan}›
‹U+1000...U+109F \p{InMyanmar}›
‹U+10A0...U+10FF \p{InGeorgian}›
‹U+1100...U+11FF \p{InHangulJamo}›
‹U+1200...U+137F \p{InEthiopic}›
‹U+1380...U+139F \p{InEthiopicSupplement}›
‹U+13A0...U+13FF \p{InCherokee}›
‹U+1400...U+167F \p{InUnifiedCanadianAboriginalSyllabics}›
‹U+1680...U+169F \p{InOgham}›
‹U+16A0...U+16FF \p{InRunic}›
‹U+1700...U+171F \p{InTagalog}›
‹U+1720...U+173F \p{InHanunoo}›
‹U+1740...U+175F \p{InBuhid}›
‹U+1760...U+177F \p{InTagbanwa}›
‹U+1780...U+17FF \p{InKhmer}›
‹U+1800...U+18AF \p{InMongolian}›
‹U+18B0...U+18FF \p{InUnifiedCanadianAboriginalSyllabicsExtended}›
‹U+1900...U+194F \p{InLimbu}›
‹U+1950...U+197F \p{InTaiLe}›
‹U+1980...U+19DF \p{InNewTaiLue}›
‹U+19E0...U+19FF \p{InKhmerSymbols}›
‹U+1A00...U+1A1F \p{InBuginese}›
‹U+1A20...U+1AAF \p{InTaiTham}›
‹U+1B00...U+1B7F \p{InBalinese}›
‹U+1B80...U+1BBF \p{InSundanese}›
‹U+1BC0...U+1BFF \p{InBatak}›
‹U+1C00...U+1C4F \p{InLepcha}›
‹U+1C50...U+1C7F \p{InOlChiki}›
‹U+1CC0...U+1CCF \p{InSundaneseSupplement}›
‹U+1CD0...U+1CFF \p{InVedicExtensions}›
‹U+1D00...U+1D7F \p{InPhoneticExtensions}›
‹U+1D80...U+1DBF \p{InPhoneticExtensionsSupplement}›
‹U+1DC0...U+1DFF \p{InCombiningDiacriticalMarksSupplement}›
‹U+1E00...U+1EFF \p{InLatinExtendedAdditional}›
‹U+1F00...U+1FFF \p{InGreekExtended}›
‹U+2000...U+206F \p{InGeneralPunctuation}›
‹U+2070...U+209F \p{InSuperscriptsandSubscripts}›
‹U+20A0...U+20CF \p{InCurrencySymbols}›
‹U+20D0...U+20FF \p{InCombiningDiacriticalMarksforSymbols}›
‹U+2100...U+214F \p{InLetterlikeSymbols}›
‹U+2150...U+218F \p{InNumberForms}›
```

```
‹U+2190...U+21FF \p{InArrows}›
‹U+2200...U+22FF \p{InMathematicalOperators}›
‹U+2300...U+23FF \p{InMiscellaneousTechnical}›
‹U+2400...U+243F \p{InControlPictures}›
‹U+2440...U+245F \p{InOpticalCharacterRecognition}›
‹U+2460...U+24FF \p{InEnclosedAlphanumerics}›
‹U+2500...U+257F \p{InBoxDrawing}›
‹U+2580...U+259F \p{InBlockElements}›
‹U+25A0...U+25FF \p{InGeometricShapes}›
‹U+2600...U+26FF \p{InMiscellaneousSymbols}›
‹U+2700...U+27BF \p{InDingbats}›
‹U+27C0...U+27EF \p{InMiscellaneousMathematicalSymbols-A}›
‹U+27F0...U+27FF \p{InSupplementalArrows-A}›
‹U+2800...U+28FF \p{InBraillePatterns}›
‹U+2900...U+297F \p{InSupplementalArrows-B}›
‹U+2980...U+29FF \p{InMiscellaneousMathematicalSymbols-B}›
‹U+2A00...U+2AFF \p{InSupplementalMathematicalOperators}›
‹U+2B00...U+2BFF \p{InMiscellaneousSymbolsandArrows}›
‹U+2C00...U+2C5F \p{InGlagolitic}›
‹U+2C60...U+2C7F \p{InLatinExtended-C}›
‹U+2C80...U+2CFF \p{InCoptic}›
‹U+2D00...U+2D2F \p{InGeorgianSupplement}›
‹U+2D30...U+2D7F \p{InTifinagh}›
‹U+2D80...U+2DDF \p{InEthiopicExtended}›
‹U+2DE0...U+2DFF \p{InCyrillicExtended-A}›
‹U+2E00...U+2E7F \p{InSupplementalPunctuation}›
‹U+2E80...U+2EFF \p{InCJKRadicalsSupplement}›
‹U+2F00...U+2FDF \p{InKangxiRadicals}›
‹U+2FF0...U+2FFF \p{InIdeographicDescriptionCharacters}›
‹U+3000...U+303F \p{InCJKSymbolsandPunctuation}›
‹U+3040...U+309F \p{InHiragana}›
‹U+30A0...U+30FF \p{InKatakana}›
‹U+3100...U+312F \p{InBopomofo}›
‹U+3130...U+318F \p{InHangulCompatibilityJamo}›
‹U+3190...U+319F \p{InKanbun}›
‹U+31A0...U+31BF \p{InBopomofoExtended}›
‹U+31C0...U+31EF \p{InCJKStrokes}›
‹U+31F0...U+31FF \p{InKatakanaPhoneticExtensions}›
‹U+3200...U+32FF \p{InEnclosedCJKLettersandMonths}›
‹U+3300...U+33FF \p{InCJKCompatibility}›
‹U+3400...U+4DBF \p{InCJKUnifiedIdeographsExtensionA}›
‹U+4DC0...U+4DFF \p{InYijingHexagramSymbols}›
‹U+4E00...U+9FFF \p{InCJKUnifiedIdeographs}›
‹U+A000...U+A48F \p{InYiSyllables}›
‹U+A490...U+A4CF \p{InYiRadicals}›
‹U+A4D0...U+A4FF \p{InLisu}›
‹U+A500...U+A63F \p{InVai}›
‹U+A640...U+A69F \p{InCyrillicExtended-B}›
‹U+A6A0...U+A6FF \p{InBamum}›
‹U+A700...U+A71F \p{InModifierToneLetters}›
‹U+A720...U+A7FF \p{InLatinExtended-D}›
```

```
‹U+A800...U+A82F \p{InSylotiNagri}›
‹U+A830...U+A83F \p{InCommonIndicNumberForms}›
‹U+A840...U+A87F \p{InPhags-pa}›
‹U+A880...U+A8DF \p{InSaurashtra}›
‹U+A8E0...U+A8FF \p{InDevanagariExtended}›
‹U+A900...U+A92F \p{InKayahLi}›
‹U+A930...U+A95F \p{InRejang}›
‹U+A960...U+A97F \p{InHangulJamoExtended-A}›
‹U+A980...U+A9DF \p{InJavanese}›
‹U+AA00...U+AA5F \p{InCham}›
‹U+AA60...U+AA7F \p{InMyanmarExtended-A}›
‹U+AA80...U+AADF \p{InTaiViet}›
‹U+AAE0...U+AAFF \p{InMeeteiMayekExtensions}›
‹U+AB00...U+AB2F \p{InEthiopicExtended-A}›
‹U+ABC0...U+ABFF \p{InMeeteiMayek}›
‹U+AC00...U+D7AF \p{InHangulSyllables}›
‹U+D7B0...U+D7FF \p{InHangulJamoExtended-B}›
‹U+D800...U+DB7F \p{InHighSurrogates}›
‹U+DB80...U+DBFF \p{InHighPrivateUseSurrogates}›
‹U+DC00...U+DFFF \p{InLowSurrogates}›
‹U+E000...U+F8FF \p{InPrivateUseArea}›
‹U+F900...U+FAFF \p{InCJKCompatibilityIdeographs}›
‹U+FB00...U+FB4F \p{InAlphabeticPresentationForms}›
‹U+FB50...U+FDFF \p{InArabicPresentationForms-A}›
‹U+FE00...U+FE0F \p{InVariationSelectors}›
‹U+FE10...U+FE1F \p{InVerticalForms}›
‹U+FE20...U+FE2F \p{InCombiningHalfMarks}›
‹U+FE30...U+FE4F \p{InCJKCompatibilityForms}›
‹U+FE50...U+FE6F \p{InSmallFormVariants}›
‹U+FE70...U+FEFF \p{InArabicPresentationForms-B}›
‹U+FF00...U+FFEF \p{InHalfwidthandFullwidthForms}›
‹U+FFF0...U+FFFF \p{InSpecials}›
```

Блок Юникода – это единый и неразрывный диапазон кодовых пунктов. Хотя многим блокам присвоены имена алфавитов и категорий Юникода, тем не менее здесь нет 100% совпадения. Название блока лишь указывает на его основное назначение.

Блок Currency **не включает в себя знаки доллара и йены. По исторически сложившимся причинам они находятся в блоках** BasicLatin **и** Latin-1 Supplement**. Но при этом оба знака попадают в категорию** Currency Symbol**. Поэтому для поиска любых знаков денежных единиц вместо комбинации** ‹\p{InCurrency}› **следует использовать** ‹\p{Sc}›**.**

Большинство блоков включают в себя кодовые пункты с неприсвоенными символами, которые попадают в категорию ‹\p{Cn}›**. Никакие другие категории Юникода и никакой алфавит не включают кодовые пункты с неприсвоенными символами.**

Комбинация ‹\p{InBlockName}› **может использоваться в диалектах .NET, XRegExp и Perl, а комбинацию** ‹\p{IsBlockName}› **надо использовать в диалекте Java.**

Диалект Perl поддерживает вариант `Is`, но лучше использовать синтаксис `In`, чтобы избежать путаницы. Для работы с алфавитами в Perl поддерживаются комбинации `‹\p{Script}›` и `‹\p{IsScript}›`, но не `‹\p{InScript}›`.

В стандарте Юникода оговаривается, что имена блоков должны быть нечувствительны к регистру символов и все различия, касающиеся пробелов, дефисов или символов подчеркивания, должны игнорироваться. К сожалению, большинство диалектов регулярных выражений не отличаются такой гибкостью. Все версии .NET и Java 4 требуют записи имен блоков в «верблюжьей» нотации, как показано в списке выше. Версии Perl 5.8 и выше, а также Java 5 и выше допускают смешивать регистр символов. Диалекты Perl, Java и .NET поддерживают также нотацию с дефисами и без пробелов, использованную в списке выше. Мы рекомендуем использовать эту нотацию. Из диалектов, обсуждаемых в этой книге, только XRegExp и Perl 5.12 (и выше) полностью соответствуют требованиям стандарта Юникода в отношении пробелов, дефисов и символов подчеркивания в именах блоков Юникода.

Алфавит Юникода

Каждый кодовый пункт, за исключением кодовых пунктов с неприсвоенными символами, является частью точно одного алфавита Юникода. Кодовые пункты с неприсвоенными символами не являются частью ни одного алфавита. Кодовые пункты со значениями до U+FFFF и с приписанными символами принадлежат следующим 72 алфавитам, определенным в версии 6.1 стандарта Юникода:

```
‹\p{Common}›                    ‹\p{Lepcha}›
‹\p{Arabic}›                    ‹\p{Limbu}›
‹\p{Armenian}›                  ‹\p{Lisu}›
‹\p{Balinese}›                  ‹\p{Malayalam}›
‹\p{Bamum}›                     ‹\p{Mandaic}›
‹\p{Batak}›                     ‹\p{Meetei_Mayek}›
‹\p{Bengali}›                   ‹\p{Mongolian}›
‹\p{Bopomofo}›                  ‹\p{Myanmar}›
‹\p{Braille}›                   ‹\p{New_Tai_Lue}›
‹\p{Buginese}›                  ‹\p{Nko}›
‹\p{Buhid}›                     ‹\p{Ogham}›
‹\p{Canadian_Aboriginal}›       ‹\p{Ol_Chiki}›
‹\p{Cham}›                      ‹\p{Oriya}›
‹\p{Cherokee}›                  ‹\p{Phags_Pa}›
‹\p{Coptic}›                    ‹\p{Rejang}›
‹\p{Cyrillic}›                  ‹\p{Runic}›
‹\p{Devanagari}›                ‹\p{Samaritan}›
‹\p{Ethiopic}›                  ‹\p{Saurashtra}›
‹\p{Georgian}›                  ‹\p{Sinhala}›
‹\p{Glagolitic}›                ‹\p{Sundanese}›
‹\p{Greek}›                     ‹\p{Syloti_Nagri}›
‹\p{Gujarati}›                  ‹\p{Syriac}›
‹\p{Gurmukhi}›                  ‹\p{Tagalog}›
```

```
‹\p{Han}›
‹\p{Hangul}›
‹\p{Hanunoo}›
‹\p{Hebrew}›
‹\p{Hiragana}›
‹\p{Inherited}›
‹\p{Javanese}›
‹\p{Kannada}›
‹\p{Katakana}›
‹\p{Kayah_Li}›
‹\p{Khmer}›
‹\p{Lao}›
‹\p{Latin}›
‹\p{Tagbanwa}›
‹\p{Tai_Le}›
‹\p{Tai_Tham}›
‹\p{Tai_Viet}›
‹\p{Tamil}›
‹\p{Telugu}›
‹\p{Thaana}›
‹\p{Thai}›
‹\p{Tibetan}›
‹\p{Tifinagh}›
‹\p{Vai}›
‹\p{Yi}›
```

Алфавит – это группа кодовых пунктов, используемых в различных системах письменности. Одни алфавиты, такие как `Thai`, применяются только в одном языке человеческого общения. Другие, такие как `Latin`, используются в нескольких языках. В некоторых языках применяется несколько алфавитов. Например, в Юникоде нет алфавита Japanese, вместо этого предлагаются алфавиты `Hiragana`, `Katakana`, `Han` и `Latin`, символы из которых обычно используются при составлении документов на японском языке.

Самым первым в списке, в нарушение алфавитного порядка следования, указан алфавит `Common`. Этот алфавит содержит все символы, общие для широкого диапазона алфавитов, такие как знаки препинания, пробел и различные вспомогательные символы.

Диалект Java требует, чтобы имена алфавитов предварялись префиксом `Is`, например `‹\p{IsYi}›`. В Perl префикс `Is` допускается, но не требуется. Диалекты XRegExp, PCRE и Ruby не допускают использование префикса `Is`.

В стандарте Юникода оговаривается, что имена алфавитов должны быть нечувствительны к регистру символов и все различия, касающиеся пробелов, дефисов или символов подчеркивания, должны игнорироваться. К сожалению, большинство диалектов регулярных выражений не отличаются такой гибкостью. Форма записи имен алфавитов в «верблюжьей» нотации и с символами подчеркивания между словами поддерживается всеми диалектами, рассматриваемыми в этой книге, которые поддерживают алфавиты Юникода.

Графема Юникода

Различия между кодовыми пунктами и символами становятся заметны, когда в игру вступают *комбинированные знаки*. Кодовому пункту U+0061 соответствует «строчная буква a латинского алфавита», а кодовому пункту U+00E0 – «строчная буква a латинского алфавита с диакритическим знаком». В комбинации они представляют то, что большинство людей назвали бы символом.

Кодовый пункт U+0300 – это диакритический знак. Он может использоваться только после буквы. Строка, состоящая из кодовых пунктов U+0061 U+0300, будет отображаться как символ à, так же как и кодовый пункт U+00E0. Комбинационный знак U+0300 отображается над символом U+0061.

Причина появления двух разных способов отображения символа с диакритическим знаком состоит в том, что уже давно существует множество кодировок символов, в которых символы с диакритическими знаками присутствуют в виде одиночных символов. Разработчики Юникода сочли, что будет полезно сохранить точное отображение устаревших наборов символов в дополнение к способу, когда диакритические знаки отделяются от базовых символов, который позволяет создавать произвольные комбинации, не поддерживаемые устаревшими кодировками.

Для вас как для пользователя регулярных выражений это особенно важно, так как все диалекты регулярных выражений, рассматриваемые в этой книге, оперируют кодовыми пунктами, а не графическими символами. Когда мы говорим, что ‹.› соответствует единственному символу, в действительности подразумевается соответствие единственному кодовому пункту. Если испытуемый текст будет состоять из двух кодовых пунктов U+0061 U+0300, который в языках программирования, таких как Java, может быть представлен строковым литералом "\u0061\u0300", точке будет соответствовать только кодовый пункт U+0061, или символ a без диакритического знака U+0300. А обоим кодовым пунктам будет соответствовать регулярное выражение ‹..›.

Диалекты Perl и PCRE предлагают специальный метасимвол ‹\X›, которому соответствует любая одиночная графема Юникода. По сути, это Юникод-версия метасимвола ‹.›. Метасимвол ‹\X› обнаружит два совпадения в тексте àà независимо от того, как он будет закодирован. Если он будет закодирован как \u00E0\u0061\u0300, первое совпадение будет соответствовать символу \u00E0, а второе – последовательности \u0061\u0300. Точка соответствует единственному кодовому пункту Юникода, и поэтому она найдет в такой строке три совпадения: \u00E0, \u0061 и \u0300.

Правила, определяющие, какие именно комбинации кодовых пунктов Юникода считаются графемами, очень сложны.[1] Вообще говоря, для поиска графем требуется обеспечить сопоставление с произвольным символом, не являющимся комбинационным знаком, и с произвольным комбинационным знаком, возможно, следующим за ним. Такое сопоставление можно обеспечить с помощью регулярного выражения ‹(?>\P{M}\p{M}*)›, допустимого во всех диалектах, поддерживающих Юникод, но не поддерживающих метасимвол ‹\X›. Комбинация ‹\P{M}› соот-

---

[1] Все эти тонкости описываются в приложении к стандарту «Unicode Standard Annex #29» по адресу *http://www.unicode.org/reports/tr29/*. Дополнительные практические рекомендации по работе с графемами Юникода можно найти в разделе «Графемы и нормализация», в главе 6 книги «Программирование на Perl, 4-е издание».

ветствует любому символу, не попадающему в категорию Mark. Комбинация ‹\p{M}*› соответствует всем следующим далее комбинационным знакам, если таковые имеются.

Мы заключили эти два метасимвола в атомарную группу, чтобы исключить возвраты при сопоставлении с ‹\p{M}*›, если следующий кодовый пункт не окажется комбинационным знаком. Выражение ‹\X{2}.› не совпадет со строкой àà, потому что после совпадения двух букв с диакритическими знаками с конструкцией ‹\X{2}› для точки ничего не останется. По той же причине не совпадет со строкой àà и выражение ‹(?>\P{M}\p{M}*){2}.›. Но выражение ‹(?:\P{M}\p{M}*){2}.› с несохраняющей группой совпадет со строкой àà, если она будет закодирована как \u00E0\u0061\u0300. Во второй итерации группы ‹\p{M}*› совпадет с \u0300. Затем сопоставление с точкой потерпит неудачу. Это вызовет возврат, что вынудит ‹\p{M}*› вернуть свое совпадение, после чего точка совпадет с \u0300.

Механизм регулярных выражений в JavaScript не поддерживает атомарную группировку. Эту особенность нельзя было добавить и в XRegExp, потому что при фактическом выполнении сопоставления XRegExp все еще опирается на механизм регулярных выражений JavaScript. Поэтому при использовании XRegExp ближайшей имитацией метасимвола ‹\X› является выражение ‹(?:\P{M}\p{M}*)›. Из-за отсутствия поддержки атомарной группировки вам придется постоянно помнить, что ‹\p{M}*› может вызвать возврат, если за ‹(?:\P{M}\p{M}*)› в регулярном выражении следует конструкция, которая может совпасть с символом из категории Mark.

Варианты

Инвертированный вариант

Метасимвол ‹\P› является обратным для метасимвола ‹\p›. Например, комбинации ‹\P{Sc}› соответствует любой символ, не имеющий свойства Юникода «Currency Symbol». Метасимвол ‹\P› поддерживается всеми диалектами, поддерживающими метасимвол ‹\p›, и применим для всех поддерживаемых категорий, блоков и алфавитов.

Символьные классы

Все диалекты, поддерживающие метасимволы ‹\u›, ‹\x›, ‹\p› и ‹\P›, допускают их использование в символьных классах. В этом случае в класс добавляется символ, представленный кодовым пунктом, или символы, принадлежащие категории, блоку или алфавиту. Например, следующему регулярному выражению будет соответствовать символ, являющийся либо открывающей кавычкой, либо закрывающей кавычкой, либо символом знака торговой марки (U+2122) :

```
[\p{Pi}\p{Pf}\u2122]
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, Ruby 1.9

```
[\p{Pi}\p{Pf}\x{2122}]
```

Параметры: нет
Диалекты: Java 7, PCRE, Perl

Перечисление всех символов

Если диалект регулярных выражений не поддерживает категории Юникода, блоки или алфавиты, можно просто перечислить символы, составляющие требуемую категорию, блок или алфавит в виде символьного класса. Для блоков это выполняется очень просто: каждый блок – это просто непрерывный диапазон, заключенный между двумя кодовыми пунктами. Например, блок Greek Extended включает символы от U+1F00 до U+1FFF:

```
[\u1F00-\u1FFF]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python, Ruby 1.9

```
[\x{1F00}-\x{1FFF}]
```

Параметры: нет
Диалекты: Java 7, PCRE, Perl

Для большинства категорий и многих алфавитов эквивалентные символьные классы будут представлять длинные списки из отдельных кодовых пунктов и коротких диапазонов. Символы, входящие в любую из категорий и во многие алфавиты, разбросаны по всей таблице Юникода. Ниже приводится аналог алфавита Greek:

```
[\u0370-\u0373\u0375-\u0377\u037A-\u037D\u0384\u0386\u0388-\u038A↵
\u038C\u038E-\u03A1\u03A3-\u03E1\u03F0-\u03FF\u1D26-\u1D2A\u1D5D-\u1D61↵
\u1D66-\u1D6A\u1DBF\u1F00-\u1F15\u1F18-\u1F1D\u1F20-\u1F45\u1F48-\u1F4D↵
\u1F50-\u1F57\u1F59\u1F5B\u1F5D\u1F5F-\u1F7D\u1F80-\u1FB4\u1FB6-\u1FC4↵
\u1FC6-\u1FD3\u1FD6-\u1FDB\u1FDD-\u1FEF\u1FF2-\u1FF4\u1FF6-\u1FFE\u2126↵
\U00010140-\U0001018A\U0001D200-\U0001D245]
```

Параметры: нет
Диалект: Python

Мы создали это регулярное выражение с помощью веб-приложения UnicodeSet, находящегося по адресу *http://unicode.org/cldr/utility/list-unicodeset.jsp*. Мы ввели `\p{Greek}` в поле ввода Input (Ввод), отметили флажки Abbreviate (Сократить) и Escape (Экранировать) и щелкнули на кнопке Show Set (Показать множество).

Только Python поддерживает этот синтаксис определения кодовых пунктов Юникода, как описывалось выше в разделе «Кодовый пункт Юникода». Чтобы это регулярное выражение можно было использовать в других диалектах, необходимо внести некоторые изменения.

Регулярное выражение будет работать во многих других диалектах, если удалить из него кодовые пункты выше U+FFFF:

```
[\u0370-\u0373\u0375-\u0377\u037A-\u037D\u0384\u0386\u0388-\u038A↵
\u038C\u038E-\u03A1\u03A3-\u03E1\u03F0-\u03FF\u1D26-\u1D2A\u1D5D-\u1D61↵
\u1D66-\u1D6A\u1DBF\u1F00-\u1F15\u1F18-\u1F1D\u1F20-\u1F45\u1F48-\u1F4D↵
\u1F50-\u1F57\u1F59\u1F5B\u1F5D\u1F5F-\u1F7D\u1F80-\u1FB4\u1FB6-\u1FC4↵
\u1FC6-\u1FD3\u1FD6-\u1FDB\u1FDD-\u1FEF\u1FF2-\u1FF4\u1FF6-\u1FFE\u2126]
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, Python, Ruby 1.9

В диалектах Perl и PCRE используется иной синтаксис определения кодовых пунктов Юникода. В оригинальном регулярном выражении следует заменить ‹\uFFFF› на ‹\x{FFFF}› и ‹\U0010FFFF› на ‹\x{10FFFF}›. Измененное регулярное выражение может использоваться также в Java 7.

```
[\x{0370}-\x{0373}\x{0375}-\x{0377}\x{037A}-\x{037D}\x{0384}\x{0386}↵
\x{0388}-\x{038A}\x{038C}\x{038E}-\x{03A1}\x{03A3}-\x{03E1}↵
\x{03F0}-\x{03FF}\x{1D26}-\x{1D2A}\x{1D5D}-\x{1D61}\x{1D66}-\x{1D6A}↵
\x{1DBF}\x{1F00}-\x{1F15}\x{1F18}-\x{1F1D}\x{1F20}-\x{1F45}↵
\x{1F48}-\x{1F4D}\x{1F50}-\x{1F57}\x{1F59}\x{1F5B}\x{1F5D}\x{1F5F}-↵
\x{1F7D}\x{1F80}-\x{1FB4}\x{1FB6}-\x{1FC4}\x{1FC6}-\x{1FD3}\x{1FD6}-↵
\x{1FDB}\x{1FDD}-\x{1FEF}\x{1FF2}-\x{1FF4}\x{1FF6}-\x{1FFE}\x{2126}↵
\x{10140}-\x{10178}\x{10179}-\x{10189}\x{1018A}\x{1D200}-\x{1D245}]
```

Параметры: нет

Диалекты: Java 7, PCRE, Perl

См. также

http://www.unicode.org – официальный веб-сайт консорциума Unicode Consortium, где можно загрузить любые официальные документы, касающиеся стандарта Юникод, таблицы символов и пр.

Юникод – это обширная тема, которой посвящены целые книги. Одна из таких книг называется «Unicode Explained», автор Джукка Корпела (Jukka K. Korpella) (O'Reilly).

Мы не можем в одном разделе описать все, что необходимо знать о кодовых пунктах Юникода, свойствах, блоках и алфавитах. Мы даже не пытаемся объяснить, зачем вам это нужно – просто вам это нужно. Комфортная простота расширенной таблицы ASCII – необитаемое место в современном глобализованном мире.

В разделах «Ограничение возможности ввода алфавитно-цифровыми символами (для любого языка)» (в рецепте 4.8) и «Ограничение числа слов» (в рецепте 4.9) описывается решение некоторых возникающих на практике задач с применением категорий Юникода.

2.8. Сопоставление с одной из нескольких альтернатив

Задача

Создать регулярное выражение, которому при многократном применении к тексту `Mary, Jane, and Sue went to Mary's house` будут соответствовать Mary, Jane, Sue и снова Mary. Последующие попытки применения должны терпеть неудачу.

Решение

```
Mary|Jane|Sue
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Символ *вертикальной черты* служит в регулярных выражениях для разделения нескольких *альтернатив*. При каждом последующем применении выражению `‹Mary|Jane|Sue›` будут соответствовать слова Mary, или Jane, или Sue. Всякий раз ему будет соответствовать только одно имя, но каждый раз новое.

Все диалекты регулярных выражений, рассматриваемые в этой книге, используют механизмы, управляемые регулярными выражениями. *Механизм* – это обычное программное обеспечение, реализующее поддержку регулярных выражений. Термин *управляемый регулярными выражениями*[1] означает, что прежде чем будет выполнен переход к следующему символу, для каждого текущего символа в испытуемом тексте будут опробованы все возможные перестановки регулярного выражения.

Когда выражение `‹Mary|Jane|Sue›` впервые применяется к тексту `Mary, Jane, and Sue went to Mary's house`, тут же обнаруживается совпадение Mary в начале строки.

Когда то же самое выражение применяется к остатку строки, то есть после щелчка на кнопке Find Next (Найти далее) в текстовом редакторе,

---

[1] Другая разновидность механизмов – это механизмы, *управляемые текстом*. Главное отличие состоит в том, что механизм, управляемый текстом, посещает каждый символ в испытуемом тексте только один раз, тогда как механизм, управляемый регулярным выражением, может посещать каждый символ несколько раз. Механизмы, управляемые текстом, обладают более высокой производительностью, но поддерживают выражения, которые являются регулярными только в математическом смысле, как описывалось в начале главы 1. Замысловатые регулярные выражения в стиле языка Perl, которые делают эту книгу такой захватывающей, могут быть реализованы только при использовании механизма, управляемого регулярными выражениями.

механизм регулярных выражений попытается сопоставить ‹Mary› с первой запятой в строке. Эта попытка завершится неудачей. Затем будет выполнена попытка найти совпадение с ‹Jane› в той же позиции, которая также закончится неудачей. Попытка найти совпадение ‹Sue› с запятой также провалится. Только после этого механизм регулярных выражений передвинется на один символ вперед. Сопоставление начнется с первого пробела, в результате поиск всех трех альтернатив также не увенчается успехом.

Когда начнется сопоставление с символа J, попытка найти соответствие первой альтернативе ‹Mary› потерпит неудачу. Затем, опять же с символа J, начнется сопоставление альтернативы ‹Jane›. В результате будет найдено соответствие Jane, и механизм регулярных выражений объявит о победе.

Обратите внимание, что соответствие Jane было найдено несмотря на то, что в испытуемом тексте имеется еще одно вхождение имени Mary, а альтернатива ‹Mary› находится в регулярном выражении перед альтернативой ‹Jane›. По крайней мере, в данном случае порядок следования альтернатив в регулярном выражении не имеет значения. Регулярное выражение находит *самое первое слева* соответствие. Оно просматривает текст слева направо, на каждом шаге пытается применить все альтернативы и останавливает попытки в первой же позиции в испытуемом тексте, где было обнаружено соответствие любой из альтернатив.

Если выполнить еще одну попытку поиска в оставшейся части строки, будет найдено соответствие Sue. Четвертая попытка поиска обнаружит еще одно соответствие Mary. Если попробовать выполнить пятую попытку поиска, она завершится неудачей, потому что ни одна из трех альтернатив не обнаружит совпадение в оставшейся части строки 's house.

Порядок следования альтернатив в регулярном выражении имеет значение, только когда две из них могут обнаружить соответствие в одной и той же позиции в строке. Например, регулярное выражение ‹Jane|Janet› имеет две альтернативы, которые совпадают с одной и той же позицией в тексте Her name is Janet. В регулярном выражении отсутствуют метасимволы границ слова. Поэтому тот факт, что ‹Jane› совпадает со словом Janet в тексте Her name is Janet только частично, не имеет никакого значения.

Выражению ‹Jane|Janet› соответствует последовательность Jane в тексте Her name is Janet, потому что механизм, управляемый регулярным выражением, отличается *нетерпеливостью*. Кроме того, что испытуемый текст просматривается слева направо в поисках первого встретившегося совпадения; альтернативы в регулярном выражении точно так же просматриваются слева направо. Механизм прекращает поиск, как только обнаруживает первую совпавшую альтернативу.

Когда выражение ‹Jane|Janet› достигает символа J в тексте Her name is Janet, обнаруживается соответствие с первой альтернативой, ‹Jane›. Вторая альтернатива даже не испытывается. Если предложить механизму

попытаться отыскать второе соответствие, в испытуемом тексте для дальнейшего просмотра останется только символ t. Поэтому не будет найдено соответствие ни с одной из альтернатив.

Существует два способа воспрепятствовать альтернативе ‹Jane› похитить соответствие у альтернативы ‹Janet›. Первый способ состоит в том, чтобы поместить более длинную альтернативу на первое место: ‹Janet|Jane›. Более надежное решение заключается в том, чтобы явно определить выполняемые действия: в данном случае выполняется поиск имен, а имена – это целые слова. Регулярные выражения не распознают слова, зато они распознают границы слов.

Так, оба выражения ‹\bJane\b|\bJanet\b› и ‹\bJanet\b|\bJane\b› совпадут со словом Janet в тексте Her name is Janet. Благодаря привязке к границам слова совпасть смогла только одна альтернатива. Порядок следования альтернатив опять оказался несущественным.

В рецепте 2.12 описывается более надежное решение: ‹\bJanet?\b›.

См. также

Рецепт 2.9, где рассказывается, как группировать части регулярных выражений. Группировка становится необходимой, когда требуется вставить несколько альтернатив в середину регулярного выражения.

2.9. Группы и сохранение части совпадения

Задача

Улучшить регулярное выражение, соответствующее именам Mary, Jane и Sue, так, чтобы найденные совпадения были целыми словами. Применить группировку, чтобы добиться этого эффекта за счет использования одной пары метасимволов границы слова во всем регулярном выражении вместо одной пары для каждой альтернативы.

Создать регулярное выражение, которому соответствовали бы любые даты в формате yyyy-mm-dd и которое сохраняет по отдельности год, месяц и день. Цель состоит в том, чтобы облегчить работу с отдельными значениями в программном коде, обрабатывающем совпадение. Будем исходить из предположения, что все даты, присутствующие в испытуемом тексте, корректны. Регулярное выражение не должно исключать такие даты, как 9999-99-99, так как они вообще не будут появляться в испытуемом тексте.

Решение

```
\b(Mary|Jane|Sue)\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b(\d\d\d\d)-(\d\d)-(\d\d)\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Оператор выбора, о котором говорилось в предыдущем разделе, имеет самый низший приоритет среди всех операторов регулярных выражений. Например, если записать выражение ‹\bMary|Jane|Sue\b›, в нем получится три альтернативы: ‹\bMary›, ‹Jane› и ‹Sue\b›. Данное регулярное выражение будет обнаруживать совпадение Jane в тексте Her name is Janet.

Чтобы в регулярном выражении что-то исключить из альтернатив, следует *сгруппировать* альтернативы. Группировка выполняется с помощью круглых скобок. Они имеют наивысший приоритет среди всех операторов регулярных выражений, как и в большинстве языков программирования. В выражении ‹\b(Mary|Jane|Sue)\b› имеется три альтернативы, ‹Mary›, ‹Jane› и ‹Sue›, заключенные в метасимволы границ слова. Это регулярное выражение не находит соответствий в тексте Her name is Janet.

Когда механизм регулярных выражений достигнет символа J в слове Janet внутри испытуемого текста, первый метасимвол границы слова обнаружит совпадение. После этого механизм перейдет к группе. Попытка сопоставления с первой альтернативой в группе, ‹Mary›, потерпит неудачу. Сопоставление со второй альтернативой, ‹Jane›, увенчается успехом. Механизм покинет группу. Теперь непроверенным в выражении остается только метасимвол ‹\b›. Сопоставление границы слова с позицией между символами e и t в конце испытуемого текста потерпит неудачу. Поэтому и вся попытка найти соответствие, начиная с позиции символа J, также потерпит неудачу.

Пара круглых скобок – это не просто группа, а *сохраняющая группа*. Для случая регулярного выражения Mary-Jane-Sue сохранение не дает никаких преимуществ, потому что оно охватывает все регулярное выражение. Возможность сохранения приобретает особое значение, когда оно охватывает только часть регулярного выражения, например ‹\b(\d\d\d\d)-(\d\d)-(\d\d)\b›.

Регулярное выражение совпадает с датами в формате yyyy-mm-dd. В точности то же самое делает регулярное выражение ‹\b\d\d\d\d-\d\d-\d\d\b›. Поскольку в этом регулярном выражении отсутствуют операторы выбора или повторения, функция группировки оказывается невостребованной, но вот функция сохранения оказывается полезной.

В регулярном выражении ‹\b(\d\d\d\d)-(\d\d)-(\d\d)\b› имеется три сохраняющих группы. Группы нумеруются порядковыми номерами открывающих круглых скобок, присваиваемыми слева направо. Подвыражение ‹(\d\d\d\d)› – это группа с номером 1. Подвыражение ‹(\d\d)› – группа с номером 2. Второе подвыражение ‹(\d\d)› – группа с номером 3.

В процессе поиска совпадений, когда механизм регулярных выражений покидает группу, достигнув закрывающей круглой скобки, он сохраняет текст, совпавший с сохраняющей группой. Когда регулярное выражение совпадает с текстом 2008-05-24, фрагмент 2008 сохраняется в первом сохранении, фрагмент 05 – во втором сохранении и фрагмент 24 – в третьем.

Существует три способа использования сохраненного текста. В рецепте 2.10 описывается, как повторно проверить соответствие сохраненного текста тому же самому регулярному выражению. В рецепте 2.21 показано, как вставлять сохраненный текст в замещающий текст при выполнении операции поиска с заменой. В рецепте 3.9, в следующей главе, описывается, как можно использовать в приложениях совпадения с частями регулярных выражений.

Варианты

Несохраняющие группы

В регулярном выражении ‹\b(Mary|Jane|Sue)\b› круглые скобки потребовались исключительно для нужд группировки. Поэтому вместо сохраняющей группировки можно было бы использовать несохраняющую группировку:

```
\b(?:Mary|Jane|Sue)\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Несохраняющая группа начинается с трех символов ‹(?:›. Закрывающая круглая скобка ‹)› закрывает ее. Несохраняющая группировка обеспечивает ту же самую возможность группировки, но без сохранения совпадений.

При подсчете открывающих круглых скобок для выяснения порядковых номеров сохраняющих групп скобки, открывающие несохраняющие группы, не учитываются. Это главное преимущество несохраняющих групп: они могут добавляться в существующие регулярные выражения, не оказывая влияния на нумерованные ссылки, указывающие на сохраняющие группы.

Еще одно преимущество несохраняющей группировки заключается в скорости работы. Если в выражении не предполагается использовать обратные ссылки на определенные группы (рецепт 2.10), вставлять сохраненные в них фрагменты в замещающий текст (рецепт 2.21) или извлекать сохраненные фрагменты в программном коде (рецепт 3.9), сохраняющая группировка просто будет давать ненужную нагрузку, которую можно ликвидировать, задействовав несохраняющую группировку. На практике выигрыш в скорости работы регулярного выражения от такой замены трудно заметить, если регулярное выражение не используется в глубоком цикле или не обрабатывает большой массив данных.

Группы с модификаторами режима

В рецепте 2.1 в описании варианта «Поиск без учета регистра символов» говорилось, что диалекты .NET, Java, PCRE, Perl и Ruby поддерживают локальные модификаторы режима, которые работают как простые переключатели: `‹sensitive(?i)caseless(?-i)sensitive›`. Хотя синтаксис модификаторов включает круглые скобки, например `‹(?i)›`, они не имеют никакого отношения к группировке.

Вместо применения переключающих модификаторов можно использовать их внутри несохраняющих групп:

```
\b(?i:Mary|Jane|Sue)\b
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
sensitive(?i:caseless)sensitive
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

При добавлении модификатора режима в несохраняющую группу режим, определяемый этим модификатором, распространяется только на часть регулярного выражения, заключенную в группу. По достижении закрывающей круглой скобки восстанавливаются предыдущие параметры режима. Так как по умолчанию поиск соответствий выполняется с учетом регистра символов, только часть выражения внутри

```
(?i:…)
```

будет нечувствительна к регистру символов.

Допускается объединять несколько модификаторов режима, например: `‹(?ism:…)›`. Для отключения режима, определяемого модификатором, следует использовать символ дефиса: выражение `‹(?-ism:…)›` отключает все три режима. Выражение `‹(?i-sm)›` включает режим нечувствительности к регистру символов (`i`) и отключает режимы «точке соответствуют границы строк» (`s`) и «символам ^ и $ соответствуют границы строк» (`m`). Эти режимы подробно описаны в рецептах 2.4 и 2.5.

См. также

Рецепт 2.10, где объясняется, как заставить регулярное выражение совпадать с тем же текстом, который совпал с сохраняющей группой.

Рецепт 2.11, где рассказывается об именованных сохраняющих группах. Возможность именования групп в регулярном выражении упрощает чтение и сопровождение регулярного выражения.

Рецепт 2.21, где описывается, как вставлять в замещающий текст фрагменты, совпавшие с сохраняющими группами, при выполнении операции поиска с заменой.

Рецепт 3.9, где объясняется, как в программном коде извлечь текст, совпавший с сохраняющей группой.

Рецепт 2.15, где рассказывается, как предотвратить бессмысленные попытки механизма регулярных выражений отыскать различные варианты совпадения с группой.

2.10. Повторный поиск соответствия с ранее совпавшим текстом

Задача

Создать регулярное выражение, совпадающее с «магическими» датами в формате yyyy-mm-dd. Магической считается дата, когда последние две цифры года, месяц и день являются одним и тем же числом. Например, магической считается дата 2008-08-08. Будем исходить из предположения, что все даты, присутствующие в испытуемом тексте, корректны. Регулярное выражение не должно исключать такие даты, как 9999-99-99, так как они вообще не будут появляться в испытуемом тексте. Требуется отыскать только магические даты.

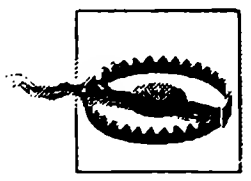
Решение

```
\b\d\d(\d\d)-\1-\1\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Чтобы в регулярном выражении повторно проверить совпадение текста, уже совпавшего ранее, для начала необходимо сохранить предыдущее совпадение. Сделать это можно с помощью сохраняющей группы, как было показано в рецепте 2.9. После этого с помощью *обратных ссылок* можно проверить совпадение с этим текстом в любом месте регулярного выражения. Ссылки на первые девять сохраняющих групп оформляются символом обратного слэша, за которым следует одиночная цифра от одного до девяти. Для групп от 10 до 99 используются формы записи от ‹\10› до ‹\99›.



Не следует использовать форму записи ‹\01›. Она будет интерпретироваться как экранированная форма записи восьмеричного числа или как ошибка. Мы вообще не будем использовать экранированные формы записи восьмеричных чисел в этой книге, потому что экранированная форма записи шестнадцатеричных чисел ‹\xFF› более наглядна.

Когда регулярное выражение ‹\b\d\d(\d\d)-\1-\1\b› встретит магическую дату, например 2008-08-08, первое подвыражение ‹\d\d› совпадет с фраг-

ментом 20. Затем механизм регулярных выражений войдет в сохраняющую группу, запомнив текущую позицию в испытуемом тексте.

Подвыражение ‹\d\d› внутри сохраняющей группы совпадет с фрагментом 08, и механизм обнаружит круглую скобку, закрывающую группу. В этой позиции совпавший фрагмент 08 будет сохранен в группе с номером 1.

Следующий элемент регулярного выражения – дефис, который совпадает с самим собой. Далее следует обратная ссылка. Механизм регулярных выражений проверит содержимое первой сохраняющей группы: 08. Затем попробует сопоставить этот текст буквально. Если регулярное выражение нечувствительно к регистру символов, сохраненный текст сопоставляется таким способом. В данном случае сопоставление с обратной ссылкой увенчается успехом. Следующий дефис и обратная ссылка также совпадут. Наконец, метасимвол границы слова совпадет с концом испытуемого текста и в качестве общего совпадения будет найдена дата 2008-08-08. Сохраняющая группа по-прежнему будет хранить фрагмент 08.

Если сохраняющая группа повторяется в результате применения квантификатора (рецепт 2.12) или возврата (рецепт 2.13), каждый раз новое совпадение с сохраняющей группой затирает предыдущее. Обратная ссылка на группу совпадает только с текстом, сохраненным этой группой.

Если то же самое выражение встретит текст 2008-05-24 2007-07-07, первый раз сохраняющая группа выполнит сохранение, когда подвыражение ‹\b\d\d(\d\d)› совпадет с фрагментом 2008, и сохранит фрагмент 08 для первой (и единственной) сохраняющей группы. Потом дефис совпадет сам с собой. Затем обратная ссылка попытается сопоставить сохраненный фрагмент ‹08› с фрагментом текста 05 и потерпит неудачу.

Поскольку в выражении отсутствуют альтернативы, механизм закончит попытку сопоставления. На этом этапе будут очищены все сохраняющие группы. Когда механизм предпримет вторую попытку сопоставления, начиная с первого символа 0 в испытуемом тексте, в группе ‹\1› вообще не будет ничего.

В следующий раз сохраняющая группа выполнит сохранение, когда подвыражение ‹\b\d\d(\d\d)› совпадет с фрагментом 2007, и сохранит фрагмент 07. Затем дефис совпадет сам с собой. Потом обратная ссылка попытается сопоставить ‹07›. На этот раз попытка увенчается успехом, как и последующие попытки сопоставления дефиса, обратной ссылки и границы слова. В результате будет найдено соответствие 2007-07-07.

Так как механизм регулярных выражений производит обработку слева направо, сохраняющие круглые скобки должны находиться перед обратной ссылкой. Регулярные выражения ‹\b\d\d\1-(\d\d)-\1› и ‹\b\d\d\1-\1-(\d\d)\b› никогда не найдут соответствие. Поскольку обратная ссылка располагается перед сохраняющей группой, в сохраняющей группе еще ничего сохранено не было. Если вы не используете JavaScript,

сопоставление обратной ссылки всегда будет приводить к неудаче, если она указывает на группу, которая еще не участвовала в сопоставлении.

Группа, не участвовавшая в сопоставлении, – это не то же самое, что группа, сохранившая совпадение нулевой длины. Обращение к обратной ссылке на группу, сохранение в которой имеет нулевую длину, всегда оканчивается успехом. Когда выражение ‹(^)\1› сопоставляется с началом строки, первая сохраняющая группа сохраняет совпадение с метасимволом ‹^›, имеющее нулевую длину; в результате попытка сопоставления с ‹\1› завершается успехом. На практике такое может случаться, когда содержимое сохраняющей группы является необязательным.

JavaScript – единственный известный нам диалект, который поступает вопреки десятилетним традициям обратных ссылок в регулярных выражениях. В диалекте JavaScript или по крайней мере в реализациях, следующих стандарту JavaScript, обратная ссылка на сохраняющую группу, не участвовавшую в сопоставлении, всегда оканчивается успехом, так же как и обратная ссылка на группу, сохранение которой имеет нулевую длину. По этой причине выражение ‹\b\d\d\1-\1-(\d\d)\b› в JavaScript может совпасть с текстом 12--34.

См. также

Рецепт 2.9, где рассказывается о сохраняющих группах в совпадениях, к которым можно обращаться с помощью обратных ссылок.

Рецепт 2.11, где описываются именованные сохраняющие группы и именованные обратные ссылки. Именованные группы и ссылки упрощают чтение и сопровождение регулярных выражений.

Рецепт 2.21, где рассказывается, как вставлять в замещающий текст фрагменты, совпавшие с сохраняющими группами, при выполнении операции поиска с заменой.

Рецепт 3.9, где объясняется, как в программном коде извлечь текст, совпавший с сохраняющей группой.

Рецепты 5.8, 5.9 и 7.11, демонстрирующие решения некоторых практических задач с использованием обратных ссылок.

2.11. Сохранение и именованные части совпадения

Задача

Создать регулярное выражение, которому соответствовали бы любые даты в формате yyyy-mm-dd и которое сохраняет по отдельности год, месяц и день. Цель состоит в том, чтобы облегчить работу с отдельными значениями в программном коде, обрабатывающем совпадение. Будем исходить из предположения, что все даты, присутствующие в испытуе-

мом тексте, корректны. Содействуя достижению этой цели, необходимо присвоить сохраняемым фрагментам текста описательные имена «year», «month» и «day».

Создать другое регулярное выражение, совпадающее с «магическими» датами в формате yyyy-mm-dd. Магической считается дата, когда последние две цифры года, месяц и день являются одним и тем же числом. Например, магической считается дата 2008-08-08. Необходимо сохранить магическое число (08 в примере) и пометить его именем «magic». Будем исходить из предположения, что все даты, присутствующие в испытуемом тексте, корректны. Регулярное выражение не должно исключать такие даты, как 9999-99-99, так как они вообще не будут появляться в испытуемом тексте.

Решение

Именованное сохранение

```
\b(?<year>\d\d\d\d)-(?<month>\d\d)-(?<day>\d\d)\b
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?'year'\d\d\d\d)-(?'month'\d\d)-(?'day'\d\d)\b
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?P<year>\d\d\d\d)-(?P<month>\d\d)-(?P<day>\d\d)\b
```

Параметры: нет
Диалекты: PCRE 4 и выше, Perl 5.10, Python

Именованные обратные ссылки

```
\b\d\d(?<magic>\d\d)-\k<magic>-\k<magic>\b
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
\b\d\d(?'magic'\d\d)-\k'magic'-\k'magic'\b
```

Параметры: нет
Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b\d\d(?P<magic>\d\d)-(?P=magic)-(?P=magic)\b
```

Параметры: нет
Диалекты: PCRE 4 и выше, Perl 5.10, Python

Обсуждение

Именованное сохранение

В рецептах 2.9 и 2.10 были продемонстрированы *сохраняющие группы* и *обратные ссылки*. Если говорить точнее, в этих рецептах использова-

лись *нумерованные* сохраняющие группы и нумерованные обратные ссылки. Каждой группе автоматически присваивается свой номер, который затем используется в обратных ссылках.

В дополнение к нумерованным группам современные диалекты регулярных выражений поддерживают *именованные* сохраняющие группы. Единственное отличие между именованными и нумерованными группами заключается в возможности присваивать описательные имена вместо того, чтобы иметь дело с автоматически присваиваемыми номерами. Именованные группы делают регулярные выражения более удобочитаемыми и простыми в сопровождении. Вставка новой сохраняющей группы в существующее регулярное выражение может изменить номера всех сохраняющих групп. Имена же, присваиваемые вами, остаются неизменными.

Python стал первым диалектом регулярных выражений, в котором появилась поддержка именованных сохранений. Синтаксис определения именованной сохраняющей группы выглядит так: `‹(?P<name>regex)›`. Имя должно состоять из символов слова, соответствующих метасимволу `‹\w›`. Комбинация `‹(?P<name>›` – это открывающая скобка группы, а `‹)›` – закрывающая.

Разработчики класса `Regex` в платформе .NET выработали собственный синтаксис определения именованных сохранений, использовав два взаимозаменяемых варианта. Синтаксис `‹(?<name>regex)›` имитирует синтаксис диалекта Python, в котором отсутствует символ `P`. Имя должно состоять из символов слова, соответствующих метасимволу `‹\w›`. Комбинация `‹(?<name>›` – это открывающая скобка группы, а `‹)›` – закрывающая.

Угловые скобки в имени сохранения здорово раздражают, когда необходимо записать выражение в документе XML, как это было при написании данной книги в формате DocBook XML. По этой причине в .NET появился альтернативный синтаксис определения именованных сохранений: `‹(?'name'regex)›`. Угловые скобки в нем были заменены апострофами. Выбирайте, какой синтаксис будет для вас удобнее. Их функциональность совершенно идентична.

Вероятно благодаря более высокой популярности .NET по сравнению с Python разработчики других библиотек поддержки регулярных выражений предпочитают воспроизводить синтаксис .NET. Этот синтаксис поддерживается в Perl 5.10 и механизмом Onigurama в Ruby 1.9. Perl 5.10 и Ruby 1.9 поддерживают оба синтаксиса – с угловыми скобками и с апострофами. Java 7 также копирует синтаксис .NET, но только вариант с угловыми скобками. Стандарт JavaScript не поддерживает именованные сохранения. Библиотека XRegExp добавляет поддержку именованных сохранений с использованием синтаксиса .NET, но она поддерживает только вариант с угловыми скобками.

В библиотеке PCRE синтаксис диалекта Python воспроизведен уже давно, когда диалект Perl вообще не поддерживал именованные сохранения. Версия PCRE 7, добавляющая новые особенности, появившиеся

в Perl 5.10, поддерживает как синтаксис диалекта .NET, так и синтаксис диалекта Python. Возможно, из-за успеха библиотеки PCRE в смысле сохранения обратной совместимости Perl 5.10 также поддерживает синтаксис диалекта Python. В PCRE и Perl 5.10 функциональность синтаксиса .NET и Python именованных сохранений полностью идентична.

Выбирайте тот синтаксис, который будет наиболее удобным для вас. Если вы используете язык PHP и стремитесь сохранить совместимость со старыми версиями PHP, включающими старые версии библиотеки PCRE, используйте синтаксис диалекта Python. Если совместимость со старыми версиями не требуется и при этом вам приходится работать с платформой .NET или Ruby, используйте синтаксис диалекта .NET, чтобы проще было переносить регулярные выражения между программами на разных языках. В случае сомнений при работе с PHP/PCRE используйте синтаксис диалекта Python. Те, кому придется пересобирать ваши программы с поддержкой более старых версий PCRE, будут недовольны, если вдруг регулярные выражения в вашем программном коде окажутся неработоспособными. При копировании регулярного выражения в программу, где используется диалект .NET или Ruby, удалить несколько лишних символов P будет совсем несложно.

Документация к PCRE 7 и Perl 5.10 лишь вскользь упоминает синтаксис диалекта Python, но это не означает, что его не следует использовать. Более того, мы рекомендуем использовать его при работе с PCRE и PHP.

Именованные обратные ссылки

В паре с именованными сохранениями идут именованные обратные ссылки. Так же как функциональность именованных сохраняющих групп идентична функциональности нумерованных сохраняющих групп, функциональность именованных обратных ссылок идентична функциональности нумерованных обратных ссылок. Зато они более удобочитаемые и проще в сопровождении.

В диалекте Python для создания обратной ссылки на группу name используется синтаксис ‹(?P=name)›. Несмотря на то что здесь используются круглые скобки, это обратная ссылка, а не группа. В этом определении нельзя ничего помещать между именем и закрывающей круглой скобкой. Обратная ссылка ‹(?P=name)› — это особый элемент регулярного выражения, так же как и ‹\1›. Диалекты PCRE и Perl 5.10 также поддерживают синтаксис именованных обратных ссылок, используемый в диалекте Python.

В диалекте .NET используются синтаксические конструкции ‹\k<name>› и ‹\k'name'›. Оба варианта совершенно идентичны по своей функциональности и являются взаимозаменяемыми. Обратная ссылка, определенная с применением апострофов, может ссылаться на именованную группу, созданную с применением угловых скобок, и наоборот. Perl 5.10, PCRE 7 и Ruby 1.9 также поддерживают синтаксис именованных об-

ратных ссылок, используемый в .NET. Java 7 и XRegExp поддерживают только вариант с угловыми скобками.

Мы настоятельно рекомендуем не смешивать именованные и нумерованные группы в одном регулярном выражении. Различные диалекты следуют разным правилам нумерации неименованных групп, находящихся между именованными группами. В Perl 5.10, Ruby 1.9, Java 7 и XRegExp используется синтаксис диалекта .NET, но они не следуют правилам нумерации, принятым в диалекте .NET для именованных сохраняющих групп или в случае смешивания нумерованных и именованных сохраняющих групп. Вместо того чтобы попытаться объяснить различия, мы просто рекомендуем не смешивать именованные и нумерованные группы. Избегайте путаницы и либо давайте имена всем неименованным группам, либо делайте их несохраняющими.

Группы с одинаковыми именами

Perl 5.10, Ruby 1.9 и .NET допускают определение нескольких сохраняющих групп с одинаковыми именами. Мы воспользуемся этой возможностью в решениях рецептов 4.5, 8.7 и 8.19. Когда в регулярном выражении используется оператор выбора для поиска различных вариантов совпадений с определенным текстом, применение сохраняющих групп с одинаковыми именами упрощает извлечение совпавших фрагментов независимо от того, с какими альтернативами они совпали. В разделе «Проверка исключительно средствами регулярного выражения» (рецепт 4.5) оператор выбора используется для сопоставления дат с учетом разной продолжительности разных месяцев. Каждая альтернатива сопоставляется с днем месяца и номером месяца. Благодаря использованию групп с одинаковыми именами – «day» и «month» – во всех альтернативах, для извлечения дня и месяца нам достаточно обратиться лишь к двум сохраняющим группам.

Все остальные диалекты, описываемые в этой книге и поддерживающие именованные сохранения, интерпретируют одинаковые имена в нескольких сохраняющих группах как ошибку.

Несколько сохраняющих групп с одинаковыми именами можно использовать, только когда в сопоставлении участвует только одна из этих групп. Это правило соблюдается во всех рецептах в данной книге, где используются сохраняющие группы с одинаковыми именами. Группы определяются в отдельных альтернативах, а альтернативы не заключены в повторяемые группы. Perl 5.10, Ruby 1.9 и .NET допускают возможность участия в сопоставлении двух групп с одинаковыми именами. Но в этом случае поведение обратных ссылок и выбор совпавшего фрагмента для сохранения несколько различаются в разных диалектах. Описать различия достаточно сложно, поэтому мы рекомендуем использовать группы с одинаковыми именами, только когда они находятся в отдельных альтернативах.

См. также

Рецепт 2.9, рассказывающий о нумерованных сохраняющих группах, в котором приводится более фундаментальная информация о работе механизма группировки в регулярных выражениях.

Рецепт 2.10, где объясняется, как заставить регулярное выражение совпадать с тем же текстом, который совпал с сохраняющей именованной группой.

Рецепт 2.11, где рассказывается об именованных сохраняющих группах. Возможность именования групп в регулярном выражении упрощает чтение и сопровождение регулярного выражения.

Рецепт 2.21, где описывается, как вставлять в замещающий текст фрагменты, совпавшие с сохраняющими группами, при выполнении операции поиска с заменой.

Рецепт 3.9, где объясняется, как в программном коде извлечь текст, совпавший с сохраняющей группой.

Рецепт 2.15, где рассказывается, как предотвратить бессмысленные попытки механизма регулярных выражений отыскать различные варианты совпадения с группой.

Во многих рецептах, что приводятся в последующих главах, именованные сохранения используются для упрощения извлечения совпадений. Некоторые из наиболее интересных решений демонстрируются в рецептах 4.5, 8.7 и 8.19.

2.12. Повторение части регулярного выражения определенное число раз

Задача

Создать регулярное выражение, которое совпадает со следующими разновидностями чисел:

- Гугол (десятичное число из единицы с сотней нулей после нее).
- 32-битное шестнадцатеричное число.
- 32-битное шестнадцатеричное число с необязательным суффиксом h.
- Вещественное число с необязательной целой частью, обязательной дробной частью и необязательной экспонентой. В каждой части допускается любое число цифр.

Решение

Гугол

```
\b\d{100}\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричное число

```
\b[a-f0-9]{1,8}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричное число с необязательным суффиксом

```
\b[a-f0-9]{1,8}h?\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Вещественное число

```
\d*\.\d+(e\d+)?
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Фиксированное число повторений

Квантификатор ‹{*n*}›, где *n* – неотрицательное целое число, повторяет предшествующий ему элемент регулярного выражения *n* раз. Подвыражение ‹\d{100}› в выражении ‹\b\d{100}\b› совпадает со строкой из 100 цифр. Тот же эффект можно получить, напечатав ‹\d› 100 раз.

Квантификатор ‹{1}› повторит предшествующий элемент один раз, как если бы квантификатора не было вообще. Выражение ‹ab{1}c› идентично выражению ‹abc›.

Квантификатор ‹{0}› повторит предшествующий элемент ноль раз, что фактически равносильно удалению его из регулярного выражения. Выражение ‹ab{0}c› идентично выражению ‹ac›.

Переменное число повторений

Для организации *повторений переменное число раз* используется квантификатор ‹{*n*,*m*}›, где *n* – неотрицательное целое число и *m* – число, большее *n*. Выражению ‹\b[a-f0-9]{1,8}\b› соответствует шестнадцатеричное число длиной от одной до восьми цифр. В случае переменного числа повторений становится существенным порядок следования альтернатив. Подробнее об этом рассказывается в рецепте 2.13.

Если числа *n* и *m* равны, выполняется фиксированное число повторений. Регулярное выражение ‹\b\d{100,100}\b› эквивалентно выражению ‹\b\d{100}\b›.

Бесконечное число повторений

Квантификатор ‹{*n*,}›, где *n* – неотрицательное целое число, позволяет выполнять *бесконечное число повторений*. По сути, бесконечное число повторений – это переменное число повторений, для которого не указан верхний предел.

Выражение ‹\d{1,}› совпадает с одной или более цифрами; то же самое делает и выражение ‹\d+›. Символ «плюс», следующий за элементом регулярного выражения, означает «один или более раз». В рецепте 2.13 описывается значение символа «плюс», следующего за квантификатором.

Выражение ‹\d{0,}› совпадает с любым числом цифр, в том числе и с нулевым; то же самое делает и выражение ‹\d*›. Символ «звездочка» всегда означает «ноль или более раз». Кроме того, что символ «звездочка» обеспечивает возможность бесконечного числа повторений, он еще делает предшествующий ему элемент необязательным.

Необязательные элементы

При использовании переменного числа повторений, когда число *n* равно нулю, элемент, предшествующий квантификатору, фактически делается необязательным. Выражение ‹h{0,1}› совпадает с ‹h› ноль или один раз. Если в испытуемом тексте отсутствует символ h, выражение ‹h{0,1}› вернет совпадение нулевой длины. Если использовать выражение ‹h{0,1}› как самостоятельное, оно будет обнаруживать совпадение нулевой длины перед каждым символом в испытуемом тексте, отличном от h. Для каждого символа h будет обнаруживаться соответствие длиной в один символ (сам символ h).

Конструкция ‹h?› – это то же самое, что и ‹h{0,1}›. Знак вопроса, следующий за допустимым элементом регулярного выражения, не являющимся квантификатором, означает «ноль или один раз». В следующем рецепте описывается значение знака вопроса, следующего за квантификатором.

Знак вопроса или любой другой квантификатор, следующий за открывающей круглой скобкой, вызывает синтаксическую ошибку. В Perl и диалектах, копирующих его, эта особенность используется для добавления «расширений Perl» в синтаксис регулярных выражений. В предыдущих рецептах демонстрируются такие особенности, как несохраняющие группы и именованные сохраняющие группы, в синтаксисе которых используется знак вопроса, следующий за открывающей круглой скобкой. Эти знаки вопроса не являются квантификаторами, они просто являются частью синтаксиса несохраняющих групп и именованных сохраняющих групп. В следующих рецептах будут показаны дополнительные способы группировки, где используется синтаксис ‹(?›.

Повторение групп

Если поместить квантификатор после круглой скобки, закрывающей группу, повторению подвергнется вся группа. Выражение ‹(?:abc){3}› идентично выражению ‹abcabcabc›.

Квантификаторы могут быть вложенными. Выражение ‹(e\d+)?› совпадет с символом e, за которым следует одна или более цифр, или обнаружит совпадение нулевой длины. В регулярном выражении, совпадающем с вещественным числом, этому подвыражению соответствует экспоненциальная часть числа.

Сохраняющие группы допускают возможность повторения. Как пояснялось в рецепте 2.9, совпадение с группой сохраняется всякий раз, когда механизм выходит за пределы группы, перезаписывая текст, совпавший с группой ранее. Выражению ‹(\d\d){1,3}› соответствует строка из двух, четырех или шести цифр. Механизм покидает группу трижды. Когда регулярное выражение совпадет со строкой 123456, сохраняющая группа будет хранить фрагмент 56, потому что фрагмент 56 был сохранен на последней итерации группы. Другие два совпадения с группой, 12 и 34, будут утрачены.

Выражение ‹(\d\d){3}› сохранит тот же текст, что и выражение ‹\d\d\d\d(\d\d)›. Если необходимо, чтобы сохраняющая группа сохраняла все две, четыре или шесть цифр, а не только две последние, вместо того чтобы повторять сохраняющую группу, следует поместить квантификатор в эту группу: ‹((?:\d\d){1,3})›. Здесь была использована несохраняющая группа, чтобы иметь возможность вынести функцию группировки из сохраняющей группы. Точно так же можно было бы использовать сохраняющие группы: ‹((\d\d){1,3})›. Когда данное выражение совпадет со строкой 123456, в ‹\1› будет храниться фрагмент 123456, а в ‹\2› – 56.

Механизм регулярных выражений платформы .NET является единственным, который позволяет извлекать все итерации повторяющейся сохраняющей группы. Если напрямую обратиться к свойству `Value` группы, которое возвращает строку, будет получено значение 56, как и в любых других механизмах регулярных выражений. Обратные ссылки в регулярном выражении и в замещающем тексте также подставят фрагмент 56, но если воспользоваться свойством `CaptureCollection` группы, можно получить доступ к стеку со значениями 56, 34 и 12.

См. также

Рецепт 2.9, где рассказывается, как группировать части регулярных выражений, чтобы их можно было повторять целиком.

Рецепт 2.13, где объясняется, как выбирать между минимальным и максимальным числом повторений.

Рецепт 2.14, где описывается, как уберечь механизм регулярных выражений от бесполезных попыток применить другое число повторений.

2.13. Выбор минимального или максимального числа повторений

Задача

Создать регулярное выражение, совпадающее с парой тегов и языка разметки XHTML и текстом между ними. Текст между этими тегами может содержать другие теги XHTML.

Решение

```
<p>.*?</p>
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Все квантификаторы, рассматривавшиеся в рецепте 2.12, являются *максимальными*, то есть они стремятся выполнить максимально возможное число повторений, возвращая часть соответствия, только если это необходимо для обеспечения совпадения всего регулярного выражения.

Это может осложнить поиск пары тегов языка разметки XHTML (который является разновидностью языка XML и потому требует, чтобы для каждого открывающего тега имелся соответствующий закрывающий тег). Рассмотрим следующий простой фрагмент текста на языке XHTML:

```
<p>
The very <em>first</em> task is to find the beginning of a paragraph.
</p>
<p>
Then you have to find the end of the paragraph
</p>
```

В этом фрагменте имеется два открывающих тега и два закрывающих . Необходимо обеспечить соответствие первого тега первому тегу , потому что они образуют единый абзац. Обратите внимание, что этот абзац содержит вложенный тег , поэтому регулярное выражение не может останавливаться, просто встретив первый символ <.

Взгляните на неправильное решение задачи из этого рецепта:

```
<p>.*</p>
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Единственное отличие этого неправильного решения заключается в отсутствии дополнительного знака вопроса после звездочки. Неправильное решение использует ту самую максимальную звездочку, что описывалась в рецепте 2.12.

После того как будет найдено соответствие с тегом в испытуемом тексте, механизм достигает конструкции ‹.*›. Точке соответствует любой символ, включая символы перевода строки. Звездочка повторяет попытки найти соответствие с точкой ноль или более раз. Звездочка является максимальным квантификатором, поэтому комбинация ‹.*› совпадает со всем, что попадается на ее пути до самого конца испытуемого текста. Повторюсь еще раз: комбинация ‹.*› «съест» весь файл XHTML, начиная от первого параграфа.

Когда ‹.*› насытится полностью, механизм попытается найти соответствие ‹<› в конце испытуемого текста. Эта попытка окончится неудачей. Но это еще не конец: механизм регулярных выражений выполнит *возврат*.

Звездочка предпочитает захватить столько текста, сколько это возможно, но ее вполне устроит и нулевое число совпадений. При каждом повторении квантификатора после превышения минимального числа повторений квантификатора регулярное выражение запоминает позицию для последующего возможного возврата. Механизм регулярных выражений может возвращаться к этим позициям, если часть регулярного выражения, следующая за квантификатором, не находит соответствия.

Когда сопоставление ‹<› терпит неудачу, механизм возвращается назад, заставляя ‹.*› вернуть один символ из своего совпадения. После этого снова предпринимается попытка найти соответствие ‹<› с последним символом в файле. И снова неудача, и снова возврат на один символ и попытка найти соответствие ‹<› с предпоследним символом в файле. Этот процесс продолжается, пока сопоставление ‹<› не увенчается успехом. Если для элемента выражения ‹<› так и не будет найдено соответствия даже после исчерпания всех возможных возвратов, то попытка найти соответствие для всего регулярного выражения будет считаться неудачной.

Если в ходе выполнения возвратов элемент ‹<› совпал в некоторой позиции, производится попытка найти соответствие элементу ‹/›. В случае неудачи механизм опять начинает выполнять возвраты. Этот процесс повторяется, пока элемент ‹</p›› не совпадет полностью.

Так в чем же собственно проблема? Так как звездочка является максимальным квантификатором, неправильному регулярному выражению будет соответствовать все содержимое файла XHTML, начиная от первого тега и заканчивая последним тегом . Но правильное решение состоит в том, чтобы отыскать абзац XHTML, то есть необходимо найти соответствие первого тега с первым тегом , следующим за ним.

В таких случаях применяются *минимальные* квантификаторы. Любой квантификатор можно сделать минимальным, поместив за ним знак вопроса: ‹*?›, ‹+?›, ‹??› и ‹{7,42}?› – все это минимальные квантификаторы.

Минимальные квантификаторы тоже позволяют выполнять возврат, но в другом режиме. Минимальный квантификатор совпадает минимально возможное число раз, сохраняет позицию возврата и позволяет механизму продолжить сопоставление остальной части регулярного

выражения. Если сопоставление с остатком регулярного выражения терпит неудачу и механизм выполняет возврат, минимальный квантификатор повторяется еще раз. Если регулярное выражение продолжает выполнять возвраты, квантификатор продолжает сопоставление, пока не будет достигнуто максимально возможное число повторений или пока повторяемый элемент не потерпит неудачу.

В выражении ‹ .*? › используется минимальный квантификатор, чтобы обеспечить корректное обнаружение единственного абзаца XHTML. После того как будет найдено совпадение с ‹ ›, минимальный квантификатор ‹.*?› просто запоминает позицию для возврата. Если подвыражение ‹ › обнаруживает совпадение сразу же вслед за , результатом сопоставления будет пустой абзац. В противном случае механизм регулярных выражений возвращается к подвыражению ‹.*?›, которому соответствует единственный символ. Если после этого сопоставление с ‹ › опять потерпит неудачу, будет выполнено сопоставление ‹.*?› со следующим символом. Эти действия будут выполняться, пока не будет обнаружено совпадение с ‹ › или ‹.*?› потерпит неудачу. Поскольку точке соответствуют любые символы, неудачное сопоставление с ‹.*?› возможно только по достижении конца файла XHTML.

Квантификаторы ‹*› и ‹*?› дают одинаковые совпадения регулярного выражения. Единственное различие между ними заключается в том, в каком порядке проверяются возможные совпадения. Максимальный квантификатор обнаружит максимально возможное соответствие. Минимальный квантификатор обнаружит минимально возможное соответствие.

Если это возможно, лучшее решение заключается в том, чтобы обеспечить существование единственно возможного соответствия. Регулярные выражения, совпадающие с числами, которые приводятся в рецепте 2.12, по-прежнему будут совпадать с теми же числами, если все имеющиеся в них квантификаторы заменить их минимальными версиями. Причина состоит в том, что части этих регулярных выражений, где имеются квантификаторы, и части, следующие за ними, являются взаимоисключающими. Метасимвол ‹\d› совпадает с цифрой, а метасимвол ‹\b›, следующий за ‹\d›, совпадает, только если следующий символ не является цифрой (или буквой).

Сравнение результатов работы выражений ‹\d+\b› и ‹\d+?\b› с различными текстами может помочь понять принцип действия минимальных и максимальных квантификаторов. Максимальная и минимальная версии воспроизводят одинаковые результаты, но испытуемый текст при этом просматривается по-разному.

В случае использования выражения ‹\d+\b› для сопоставления с текстом 1234 элемент ‹\d+› совпадет со всеми цифрами. После этого будет обнаружено совпадение с метасимволом ‹\b›, и в результате будет найдено общее совпадение с выражением. В случае использования выражения ‹\d+?\b›, элемент ‹\d+?› сначала совпадет только с цифрой 1. Сопоставление

метасимвола ‹\b› с позицией между 1 и 2 потерпит неудачу. Затем соответствие с ‹\d+?› будет расширено до 12, и снова сопоставление ‹\b› потерпит неудачу. Так будет продолжаться, пока ‹\d+?› не совпадет с 1234, после чего успешно будет обнаружено совпадение с ‹\b›.

Если в качестве испытуемого текста взять 1234X, первое регулярное выражение, ‹\d+\b›, по-прежнему будет совпадать с фрагментом 1234. Но сопоставление с ‹\b› потерпит неудачу. Подвыражение ‹\d+› вернет одно совпадение, оставив 123. Сопоставление с ‹\b› снова потерпит неудачу. Подвыражение ‹\d+› будет возвращать совпадения, пока не останется его минимум, то есть 1, но и в этом случае сопоставление с ‹\b› будет терпеть неудачу. В результате попытка найти совпадение для всего выражения будет считаться неудачной.

При применении выражения ‹\d+?\b› к тексту 1234X подвыражение ‹\d+?› сначала совпадет только с 1. Сопоставление метасимвола ‹\b› с позицией между 1 и 2 потерпит неудачу. Затем соответствие с ‹\d+?› будет расширено до 12, и снова сопоставление ‹\b› потерпит неудачу. Так будет продолжаться, пока ‹\d+?› не совпадет с 1234, но и после этого сопоставление с ‹\b› будет терпеть неудачу. Подвыражение ‹\d+?› попытается расширить соответствие, но сопоставление подвыражения ‹\d› с символом X потерпит неудачу. В результате попытка найти совпадение для всего выражения будет считаться неудачной.

Когда между границами слова помещается подвыражение ‹\d+›, оно должно будет либо совпасть со всеми цифрами в испытуемом тексте, либо потерпеть неудачу. Использование минимальной версии квантификатора не изменит результирующего совпадения или неудачи. Фактически использование выражения ‹\b\d+\b› было бы более предпочтительно, если бы оно могло обнаруживать совпадение вообще без возвратов. В следующем рецепте описывается, как для достижения этого эффекта можно использовать захватывающий квантификатор ‹\b\d++\b›, по крайней мере в некоторых диалектах.

См. также

Рецепт 2.8, где описывается, как механизм регулярных выражений пытается применять альтернативы при использовании оператора выбора. Это еще одна форма возвратов.

Рецепт 2.12, где демонстрируются другие операторы выбора, поддерживаемые регулярными выражениями.

Рецепт 2.9, где рассказывается, как группировать части регулярных выражений, чтобы их можно было повторять целиком.

Рецепт 2.14, где описывается, как уберечь механизм регулярных выражений от бесполезных попыток применить другое число повторений.

Рецепт 2.15, где рассказывается, как предотвратить бессмысленные попытки механизма регулярных выражений отыскать различные варианты совпадения с группой.

2.14. Устранение бесполезных возвратов

Задача

В предыдущем рецепте описывались различия между максимальными и минимальными квантификаторами и как они выполняют возврат. В некоторых ситуациях в возвратах нет никакой необходимости.

В выражении ‹\b\d+\b› используется максимальный квантификатор, а в выражении ‹\b\d+?\b› – минимальный. Оба они совпадают с целым числом. В одном и том же тексте они обнаруживают одно и то же совпадение. Любые выполняемые ими возвраты не являются необходимыми. Нужно переписать это регулярное выражение и явно ликвидировать все возвраты, что обеспечить более высокую производительность.

Решение

```
\b\d++\b
```

Параметры: нет
Диалекты: Java, PCRE, Perl 5.10, Ruby 1.9

Простейшее решение заключается в использовании захватывающего квантификатора. Но такие квантификаторы поддерживаются не всеми современными диалектами регулярных выражений.

```
\b(?>\d+)\b
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

Атомарная группировка оказывает тот же эффект. Она имеет менее удобочитаемый синтаксис, но поддерживается более широким кругом диалектов, чем захватывающие квантификаторы.

JavaScript и Python не поддерживают ни захватывающие квантификаторы, ни атомарную группировку. В этих двух диалектах нет никакой возможности ликвидировать бесполезные возвраты.

Обсуждение

Захватывающий квантификатор напоминает максимальный квантификатор: он пытается отыскать максимально возможное число соответствий. Различие состоит в том, что захватывающий квантификатор не возвращает совпадения, даже когда возврат является единственным способом, который мог бы обеспечить совпадение для остальной части выражения. Захватывающие квантификаторы не запоминают позиции совпадений для возврата.

Любой квантификатор можно сделать захватывающим, поместив вслед за ним символ «плюс». Например, все следующие квантификаторы являются захватывающими: ‹*+›, ‹++›, ‹?+› и ‹{7,42}+›.

Захватывающие квантификаторы поддерживаются Java 4 и выше, с момента включения пакета `java.util.regex`. Все версии PCRE, рассматриваемые в этой книге (с 4 по 7), поддерживают захватывающие квантификаторы. Диалект Perl поддерживает их, начиная с версии Perl 5.10. Классический механизм регулярных выражений Ruby не поддерживает захватывающие квантификаторы, а механизм Onigurama, по умолчанию используемый в Ruby 1.9, обеспечивает их поддержку.

Обертывание максимального квантификатора *атомарной группировкой* дает тот же эффект, что и захватывающий квантификатор. Когда механизм регулярных выражений покидает атомарную группу, все позиции возврата, сохраненные квантификатором, и варианты выбора внутри группы отбрасываются. Атомарная группировка имеет синтаксис: ‹(?>···)›, где ‹···› – любое регулярное выражение. Атомарная группа по своей сути является несохраняющей группой, которая дополнительно отвергает возвраты. Знак вопроса не является квантификатором; просто открывающая скобка состоит из трех символов ‹(?>›.

В случае применения выражения ‹\b\d++\b› (с захватывающим квантификатором) к тексту `123abc 456` метасимвол ‹\b› совпадет с началом испытуемого текста, а подвыражение ‹\d++› совпадет с фрагментом `123`. В этом оно не отличается от выражения ‹\b\d+\b› (с максимальным квантификатором). Но затем при сопоставлении с позицией между `3` и `a` второй метасимвол ‹\b› потерпит неудачу.

Захватывающий квантификатор не сохраняет позиции возвратов. Поскольку в этом выражении отсутствуют другие квантификаторы или варианты выбора, у механизма не остается возможности попробовать найти иные соответствия, когда сопоставление со второй границей слова терпит неудачу. В результате механизм тут же объявляет о неудаче поиска соответствия в позиции символа `1`.

Механизм попытается сопоставить регулярное выражение, начиная со следующего символа в строке, и использование захватывающего квантификатора не повлияет на такое его поведение. Если необходимо обеспечить совпадение регулярного выражения со всем испытуемым текстом, следует использовать якорные метасимволы, как обсуждалось в рецепте 2.5. В конечном счете механизм регулярных выражений дойдет до попытки сопоставить выражение, начиная с позиции символа `4`, и обнаружит соответствие `456`.

Отличие выражения с максимальным квантификатором состоит в том, что в случае первой неудачной попытки сопоставить метасимвол ‹\b› максимальный квантификатор уступит одно свое соответствие. После этого механизм проверит (без всякой пользы) совпадение ‹\b› с позицией между символами `2` и `3`, а затем между символами `1` и `2`.

Процесс поиска совпадения с применением атомарной группировки в сущности выполняется точно так же. При применении выражения ‹\b(?>\d+)\b› (захватывающего) к тексту `123abc 456` граница слова совпадет с началом испытуемого текста. Механизм регулярных выражений

войдет в атомарную группу, и подвыражение ‹\d+› совпадет с фрагментом 123. Затем механизм выйдет из атомарной группы, и все позиции возврата, сохраненные подвыражением ‹\d+›, будут утрачены. Когда сопоставление со вторым метасимволом ‹\b› потерпит неудачу, у механизма регулярных выражений не останется вариантов, как сразу же объявить о неудачной попытке. Как и при использовании захватывающего квантификатора, в конечном итоге это выражение обнаружит соответствие 456.

Мы говорили, что захватывающий квантификатор не сохраняет позиции возврата, а атомарная группировка отбрасывает их. Это позволяет лучше понять процесс сопоставления, но не стоит заострять свое внимание на различиях, так как в некоторых диалектах их вообще может не быть. Во многих диалектах ‹x++› – это всего лишь сокращенная форма записи ‹(?>x+)›, и оба выражения работают совершенно одинаково. Для конечного результата совершенно неважно, сразу ли механизм не запоминает позиции возврата или отбрасывает их позднее.

Основные же различия между захватывающими квантификаторами и атомарной группировкой заключаются в том, что захватывающий квантификатор применяется к единственному элементу регулярного выражения, тогда как атомарная группа может заключать в себя целое регулярное выражение.

Выражения ‹\w++\d++› и ‹(?>\w+\d+)› – это не одно и то же. Выражение ‹\w++\d++›, эквивалентом которого является выражение ‹(?>\w+)(?>\d+)›, не совпадет с текстом abc123. Подвыражение ‹\w++› совпадет со всем текстом abc123, но затем механизм регулярных выражений попытается найти в конце испытуемого текста соответствие для ‹\d++›. Поскольку в тексте не останется символов, которые могли бы соответствовать, подвыражение ‹\d++› потерпит неудачу. В отсутствие позиций возврата попытка сопоставить регулярное выражение провалится.

В выражении ‹(?>\w+\d+)› используются два максимальных квантификатора, заключенные в такую же атомарную группу. Внутри этой группы возвраты выполняются как обычно. Позиции возвратов отбрасываются только после того, как механизм регулярных выражений покинет группу. При применении этого выражения к тексту abc123 подвыражение ‹\w+› совпадет с фрагментом abc123. Максимальные квантификаторы запоминают позиции возвратов. Когда сопоставление с ‹\d+› потерпит неудачу, подвыражение ‹\w+› уступит один символ. В результате для ‹\d+› будет найдено соответствие 3. После этого механизм регулярных выражений покинет группу и отбросит все позиции возврата, сохраненные подвыражениями ‹\w+› и ‹\d+›. Поскольку был достигнут конец регулярного выражения, различия становятся несущественными. Общее совпадение с регулярным выражением обнаружено.

Если бы конец регулярного выражения не был достигнут, как, например, при использовании выражения ‹(?>\w+\d+)\d+›, механизм регулярных выражений оказался бы в той же ситуации, что и в случае с выра-

жением ‹\w++\d++›. Для второго подвыражения ‹\d+› не остается символов для сопоставления в конце испытуемого текста. А так как позиции возврата уже были отброшены, механизму регулярных выражений остается только объявить о неудаче.

Захватывающие квантификаторы и атомарная группировка не только оптимизируют регулярные выражения. Они могут оказывать влияние на соответствия, обнаруживаемые регулярным выражением, устраняя результаты, которые могли бы быть получены за счет возвратов.

Этот рецепт продемонстрировал, что применение захватывающих квантификаторов и атомарной группировки позволяет выполнить минимальную оптимизацию, которая может даже не повлиять на результаты тестов на скорость выполнения. В следующем рецепте будет показано, что атомарная группировка может оказывать весьма существенное влияние.

См. также

Рецепт 2.12, где демонстрируются другие операторы выбора, поддерживаемые регулярными выражениями.

Рецепт 2.15, где рассказывается, как предотвратить бессмысленные попытки механизма регулярных выражений отыскать различные варианты совпадения с группой.

2.15. Предотвращение бесконтрольных повторений

Задача

Создать единственное регулярное выражение, совпадающее с целым файлом HTML, которое будет проверять наличие тегов `html`, `head`, `title` и `body` и их вложенность. Выражение не должно совпадать с файлами HTML, в которых отсутствуют требуемые теги.

Решение

```
<html>(?>.*?<head>)(?>.*?<title>)(?>.*?</title>)↵
(?>.*?</head>)(?>.*?<body[^>]*>)(?>.*?</body>).*?</html>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Ruby

Диалекты JavaScript и Python не поддерживают атомарную группировку. В них нет никакой возможности устранить бесполезные возвраты. При программировании на JavaScript или Python эту проблему можно решить за счет поиска литералов тегов по одному, выполняя поиск следующего тега по остатку испытуемого текста, после того как будет найден предыдущий.

Обсуждение

Правильное решение этой задачи будет проще понять, если начать с более простого решения:

```
<html>.*?<head>.*?<title>.*?</title>↵
.*?</head>.*?<body[^>]*>.*?</body>.*?</html>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если проверить работу этого выражения, применив его к корректному файлу HTML, оно прекрасно справится со своей задачей. Подвыражение `<.*?>` пропустит все, поэтому включен параметр «точке соответствуют границы строк». Минимальная звездочка гарантирует, что регулярное выражение будет продвигаться вперед по одному символу за раз, каждый раз проверяя совпадение со следующим тегом. Все это объясняется в рецептах 2.4 и 2.13.

Однако проблемы возникнут, если применить это выражение к испытуемому тексту, в котором отсутствуют какие-либо теги HTML. Самый худший случай – отсутствие тега `</html>`.

Предположим, что механизм регулярных выражений обнаружил соответствия со всеми предыдущими тегами и теперь занят разворачиванием последнего подвыражения `<.*?>`. Поскольку подвыражение `<</html>>` никогда не обнаружит соответствие, совпадение с `<.*?>` будет расширяться вплоть до конца файла. Когда это произойдет, попытка найти совпадение со всем регулярным выражением будет признана неудачной.

Но это еще не конец. Другие шесть подвыражений `<.*?>` имеют сохраненные позиции возврата, которые позволяют расширять их соответствия. Когда последнее подвыражение `<.*?>` потерпит неудачу, расширяться начнет предшествующее ему, постепенно захватывая совпадение `</body>`. Тот самый текст, который ранее уже был определен как совпадение с литералом `<</body>>` в регулярном выражении. Это подвыражение `<.*?>` тоже будет расширено до самого конца файла, как и все предыдущие точки с минимальным квантификатором. Только когда первое подвыражение `<.*?>` достигнет конца файла, механизм регулярных выражений объявит о неудаче.

Это регулярное выражение имеет сложность[1] $O(n^7)$ для самого тяжелого случая, то есть длина испытуемого текста в седьмой степени. В выражении имеется семь точек с минимальным квантификатором, потенци-

---

[1] Сложность компьютерных алгоритмов обычно описывается с применением «нотации большого О». Отличный обзор способов оценки сложности компьютерных алгоритмов можно найти в статье *http://en.wikipedia.org/wiki/Time_complexity*. (В русскоязычной Википедии нет подобной статьи, однако для общего знакомства можно порекомендовать статью *http://habrahabr.ru/post/104219/*. – *Прим. перев.*)

ально способных пройти попытки сопоставления до конца файла. Если размер файла увеличится в два раза, регулярному выражению может понадобиться до 128 раз больше шагов, чтобы обнаружить отсутствие совпадения.

Мы называем это *катастрофическими возвратами*. Такое большое число возвратов может привести к тому, что регулярное выражение навечно «застрянет» в процедуре поиска или вызовет аварийное завершение приложения. Некоторые реализации регулярных выражений способны прервать неконтролируемую последовательность повторений раньше, но даже в этом случае регулярное выражение будет способно до минимума снизить производительность приложения.

Катастрофические возвраты – это пример класса явлений, известного как *комбинаторный взрыв*, когда пересекаются несколько независимых условий и необходимо проверить все возможные комбинации. Можно даже сказать, что в этом случае регулярное выражение – это *декартово произведение* различных операторов повторения.

Решение этой проблемы заключается в использовании атомарной группировки, предотвращающей бесполезные возвраты. Нет никакого смысла расширять соответствие для шестого подвыражения ‹.*?›, после того как будет найдено совпадение для ‹</body>›. Если попытка найти совпадение с ‹</html>› не увенчалась успехом, расширение соответствия для шестой точки с минимальным квантификатором не приведет к появлению закрывающего тега `html`.

Чтобы остановить расширение соответствия для квантифицированного элемента регулярного выражения, когда было найдено совпадение со следующим за ним разделителем, необходимо квантифицированную часть выражения и разделитель поместить в атомарную группу: ‹(?>.*?</body>)›. Теперь механизм регулярных выражений будет отбрасывать все позиции, совпавшие с ‹.*?</body>›, после того как будет найдено соответствие для ‹</body>›. Если позднее сопоставление с ‹</html>› потерпит неудачу, механизм регулярных выражений забудет о существовании ‹.*?</body>› и дальнейшее расширение соответствия происходить не будет.

Если точно так же оформить все остальные подвыражения ‹.*?›, ни одно из них не будет расширять свое соответствие. Хотя в выражении по-прежнему останутся все семь точек с минимальными квантификаторами, их совпадения никогда больше не будут перекрываться. Это снизит сложность регулярного выражения до O(n), которая имеет линейную зависимость от длины испытуемого текста. Никакое регулярное выражение не может быть более эффективным, чем это.

Варианты

При желании катастрофическое действие возвратов можно наблюдать, применив регулярное выражение ‹(x+x+)+y› к тексту xxxxxxxxxx. Если

оно потерпит неудачу слишком быстро, добавьте в испытуемый текст еще один символ x. Повторяйте добавлять символы, пока регулярное выражение не будет выполняться слишком долго или пока приложение не завершится аварийно. Для этого не потребуется добавлять слишком много символов x, если не использовать Perl для тестирования.

Из всех диалектов регулярных выражений, обсуждаемых в этой книге, только Perl способен определить, что выражение слишком сложное, и прервать попытки поиска соответствий без аварийного завершения приложения.

Это регулярное выражение имеет сложность $O(2^n)$. Когда попытка найти соответствие для ‹y› потерпит неудачу, механизм регулярных выражений опробует все возможные перестановки повторений для каждого элемента ‹x+› и группы, содержащей их. Например, одна такая перестановка, возникающая в процессе сопоставления, – это когда первому подвыражению ‹x+› соответствует фрагмент xxx, второму подвыражению ‹x+› – символ x, и группа повторяется три или более раз для каждого совпадения ‹x+› с x. Для текста с десятью символами x будет выполнено 1024 перестановки. Если увеличить число символов в испытуемом тексте до 32, будет получено более 4 миллиардов возможных перестановок, что наверняка приведет к исчерпанию доступной памяти механизмом регулярных выражений, если в нем отсутствует «аварийная кнопка», которая позволит ему сдаться и заявить, что регулярное выражение слишком сложное.

В данном случае бессмысленное регулярное выражение легко можно записать как ‹xx+y›, которое будет отыскивать те же совпадения, но зависимость сложности от длины текста будет уже линейной. На практике в случае более сложных регулярных выражений правильные решения могут быть не такими очевидными.

Суть состоит в том, чтобы научиться видеть ситуации, когда две или более частей регулярного выражения могут совпасть с одним и тем же текстом. В таких ситуациях может потребоваться применить атомарную группировку, чтобы исключить для механизма регулярных выражений возможность попытаться разделить испытуемый текст между этими частями выражения всеми возможными способами. Регулярные выражения всегда следует проверять на (длинных) испытуемых текстах, содержащих фрагменты, частично, но не полностью соответствующие регулярному выражению.

См. также

Рецепт 2.13, где объясняется, как выбирать между минимальным и максимальным числом повторений.

Рецепт 2.14, где описывается, как уберечь механизм регулярных выражений от бесполезных попыток применить другое число повторений.

Раздел «Графема в Юникоде» (рецепт 2.7), где приводится другой пример применения атомарной группировки для предотвращения нежелательных совпадений.

Раздел «SDL Regex Fuzzer» (глава 1), где описывается приложение SDL Regex Fuzzer, способное проверять (некоторые) регулярные выражения на подверженность проблеме катастрофических возвратов.

2.16. Проверка соответствия без включения его в общее соответствие

Задача

Отыскать любое слово, расположенное между парой тегов `<b>` и `</b>` HTML, без включения этих тегов в общее соответствие регулярному выражению. Например, для испытуемого текста `My <b>cat</b> is furry` правильным соответствием будет `cat`.

Решение

```
(?<=<b>)\w+(?=</b>)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 поддерживают опережающую проверку `‹(?=</b>)›`, но не поддерживают ретроспективную `‹(?<=</b>)›`.

Обсуждение

Проверка соседних символов

Четыре вида групп, выполняющих *проверку соседних символов* и поддерживаемых современными диалектами регулярных выражений, имеют специальный механизм отказа от текста, совпавшего с частью регулярного выражения внутри таких групп. По сути, проверка соседних символов позволяет выяснить, совпадает ли близлежащий текст, не помещая его в совпадение.

Проверка предыдущих символов называется *ретроспективной*. Это единственная конструкция в регулярных выражениях, которая просматривает текст справа налево, а не слева направо. *Положительная ретроспективная проверка* имеет синтаксис `‹(?<=···)›`. Четыре символа `‹(?<=›` образуют открывающую скобку. Что допускается включать в содержимое ретроспективной проверки, показанное здесь как `‹···›`, зависит от используемого диалекта регулярных выражений. Но как бы то ни было, простые строковые литералы, такие как `‹(?<=<b>)›`, можно использовать в любом диалекте.

Ретроспективная проверка исследует, совпадает ли текст, указанный в ней, с текстом, расположенным непосредственно перед позицией, где

механизм регулярных выражений встретил ретроспективную проверку. Применительно к тексту `My <b>cat</b> is furry` ретроспективная проверка `‹(?<=<b>)›` будет терпеть неудачу, пока не будет выполнена попытка сопоставления в позиции символа `c`. Тогда механизм регулярных выражений войдет в группу ретроспективной проверки, предложив ей проверить текст слева. Элемент `‹<b>›` совпадет с текстом левее символа `c`. Механизм покинет ретроспективную проверку в этой позиции и исключит текст, совпавший с ретроспективной проверкой, из общего совпадения. Говоря другими словами, общее соответствие возвращается к состоянию, которое было на момент, когда механизм вошел в ретроспективную проверку. В данном случае перед попыткой сопоставления с символом `c` в испытуемой строке текущее совпадение имело нулевую длину. Ретроспективная проверка лишь проверяет наличие соответствия с `‹<b>›`. При этом текст, совпавший с ней, в общее совпадение не включается. По этой причине проверки соседних символов называют еще *проверками с нулевой длиной совпадения*.

После совпадения с ретроспективной проверкой предпринимается попытка сопоставить символьный класс `‹\w+›` с одним или более символами. Он совпадет со словом `cat`. Подвыражение `‹\w+›` не находится внутри какой-либо проверки соседних символов или группы, поэтому оно совпадет с текстом `cat` обычным способом. Мы говорим, что `‹\w+›` совпадает и *поглощает* текст `cat`, тогда как проверка соседних символов может совпадать, но никогда ничего не поглощает.

Проверка может также выполняться в прямом направлении, в каком регулярное выражение просматривает текст. Такая проверка называется *опережающей*. Опережающая проверка в равной степени поддерживается всеми диалектами регулярных выражений, рассматриваемыми в этой книге. *Положительная опережающая проверка* имеет синтаксис `‹(?=···)›`. Первые три символа `‹(?=›` образуют открывающую скобку группы. Все, что допускается использовать в регулярных выражениях, можно использовать внутри опережающей проверки вместо `‹···›`.

Когда подвыражение `‹\w+›` в выражении `‹(?<=<b>)\w+(?=</b>)›` совпадет с фрагментом `cat` в тексте `My <b>cat</b> is furry`, механизм регулярных выражений войдет в опережающую проверку. Единственная особенность поведения опережающей проверки в этой точке заключается в том, что механизм регулярных выражений запоминает, какая часть текста уже совпала, связывая ее с опережающей проверкой. Совпадение с `‹</b>›` будет обнаружено как обычно, и механизм регулярных выражений покинет опережающую проверку. Регулярное выражение внутри опережающей проверки обнаружило совпадение, поэтому и сама опережающая проверка считается совпавшей. Механизм регулярных выражений отбросит текст, совпавший с опережающей проверкой, и восстановит соответствие в состояние, существовавшее на момент входа в опережающую проверку. Общее текущее совпадение будет содержать текст `cat`. Поскольку на этом регулярное выражение заканчивается, в качестве окончательного результата поиска соответствия принимается текст `cat`.

Отрицательная проверка

Форма записи ‹(?!···)› с восклицательным знаком вместо знака равенства – это *отрицательная опережающая проверка*. Отрицательная опережающая проверка работает точно так же, как и положительная, за исключением того, что положительная опережающая проверка совпадает, когда совпадает регулярное выражение, находящееся внутри нее, а отрицательная опережающая проверка совпадает, когда регулярное выражение внутри нее терпит неудачу.

Процесс сопоставления протекает точно так же. Механизм сохраняет текущее найденное совпадение при входе в отрицательную опережающую проверку и пытается сопоставить регулярное выражение внутри нее обычным способом. Если подвыражение совпадает, опережающая проверка терпит неудачу и механизм регулярных выражений выполняет возврат. Если совпадение с подвыражением обнаружено не будет, механизм восстанавливает прежнее соответствие и продолжает сопоставление оставшейся части регулярного выражения.

Точно так же форма записи ‹(?<!···)› является *отрицательной ретроспективной проверкой*. Отрицательная ретроспективная проверка совпадает, когда ни одна из альтернатив внутри ретроспективной проверки не обнаруживает совпадение с текстом, предшествующим позиции в испытуемом тексте, которая была достигнута механизмом регулярных выражений.

Различные уровни ретроспективной проверки

Опережающая проверка выполняется довольно просто. Все диалекты регулярных выражений, рассматриваемые в этой книге, позволяют помещать в нее полноценные регулярные выражения. Все конструкции, какие только можно использовать в регулярном выражении, можно использовать и в опережающей проверке. Внутрь опережающей проверки можно даже вкладывать другие опережающие и ретроспективные проверки. От таких нагромождений голова может пойти кругом, но механизм регулярных выражений справится с этим без труда.

Ретроспективная проверка – это совсем другая история. Программное обеспечение поддержки регулярных выражений всегда предназначалось для поиска по тексту только в направлении слева направо. Ретроспективный поиск часто реализуется довольно грубо: механизм регулярных выражений определяет количество символов, помещенных внутрь ретроспективной проверки, переходит назад на указанное число символов и затем сравнивает текст в ретроспективной проверке с испытуемым текстом слева направо.

По этой причине первые реализации допускали вставлять внутрь ретроспективной проверки только текст фиксированной длины. Диалекты Perl и Python все еще требуют от ретроспективной проверки, чтобы она имела фиксированную длину, но допускают включение элементов фик-

сированной длины, таких как символьные классы и операторы выбора, при условии, что в последних все альтернативы будут содержать одинаковое количество символов.

Разработчики PCRE и Ruby 1.9 пошли на один шаг вперед, позволив использовать внутри ретроспективных проверок операторы выбора с альтернативами разной длины при условии, что каждая альтернатива имеет фиксированную длину. Они способны обрабатывать регулярные выражения, такие как `(?<=one|two|three|forty-two|gr[ae]y)`, но не сложнее. Внутренние реализации диалектов PCRE и Ruby 1.9 разложат это выражение на шесть ретроспективных проверок. Сначала они вернутся назад на три символа, чтобы проверить варианты `one|two`, затем на четыре символа, чтобы проверить варианты `gray|grey`, затем на пять символов, чтобы проверить `three`, и наконец, на девять символов, чтобы проверить `forty-two`.

Разработчики диалекта Java пошли еще на один шаг вперед. В этом диалекте допускается использовать внутри ретроспективных проверок любые регулярные выражения, имеющие конечную длину совпадения. Это означает, что внутри ретроспективной проверки можно использовать все, за исключением бесконечных квантификаторов `*`, `+` и `{42,}`. Внутренняя реализация диалекта Java вычисляет минимальную и максимальную длину текста, который возможно будет соответствовать регулярному выражению внутри ретроспективной проверки. Затем она возвращается на минимальное число символов и применяет регулярное выражение из ретроспективной проверки в направлении слева направо. Если совпадение не будет обнаружено, она возвращается еще на один символ назад и повторяет попытку, пока либо ретроспективная проверка не совпадет, либо не будет выполнена попытка сопоставить максимальное число символов.

Если все это на ваш взгляд выглядит неэффективным, пусть так. Ретроспективная проверка удобна в использовании, но с ее применением невозможно поставить рекорд в скорости выполнения. Ниже мы представим решение для диалектов JavaScript и Ruby 1.8, которые вообще не поддерживают ретроспективные проверки. В действительности же это решение намного более эффективно, чем решение на основе применения ретроспективной проверки.

Механизм регулярных выражений платформы .NET является единственным в мире[1], который фактически способен обрабатывать полноценные регулярные выражения внутри ретроспективной проверки – в действительности, он применяет такие регулярные выражения в направ-

---

[1] Механизм регулярных выражений, реализованный в программе RegexBuddy, также позволяет помещать полноценные регулярные выражения внутрь ретроспективной проверки, но в нем (пока) отсутствует особенность, которая напоминала бы свойство `RegexOptions.RightToLeft`, имеющееся в реализации платформы .NET, используемое для переворачивания всего регулярного выражения.

лении справа налево. И само регулярное выражение в ретроспективной проверке, и испытуемый текст просматриваются справа налево.

Двойная проверка на совпадение с одним и тем же текстом

При использовании ретроспективной проверки в начале регулярного выражения или опережающей проверки в конце возникает ситуация, когда необходимо, чтобы что-то находилось перед или после совпадения с регулярным выражением без включения этого «что-то» в общее совпадение. Если проверка соседних символов выполняется в середине регулярного выражения, можно применять несколько проверок на совпадение с одним и тем же текстом.

В рецепте 2.3 в разделе «Особенности, характерные для разных диалектов» было показано, как использовать разность символьных классов для поиска цифр алфавита Thai. Операцию вычитания символьных классов поддерживают только .NET и Java.

Символ является тайской цифрой, если он является символом (любым) алфавита Thai и цифрой (любого алфавита). С помощью опережающей проверки можно проверить соответствие обоим требованиям для одного и того же символа:

```
(?=\p{Thai})\p{N}
```

Параметры: нет
Диалекты: PCRE, Perl, Ruby 1.9

Это регулярное выражение работает только в трех диалектах, поддерживающих алфавиты Юникода, как уже описывалось в рецепте 2.7. Но сам принцип использования опережающей проверки для сопоставления одного и того же символа более одного раза может использоваться во всех диалектах, описываемых в этой книге.

Когда механизм регулярных выражений выполняет поиск совпадения для выражения ‹(?=\p{Thai})\p{N}›, он начинает с того, что входит в опережающую проверку для каждой позиции в испытуемой строке, откуда начинается попытка сопоставления. Если символ в данной позиции отсутствует в алфавите Thai (то есть проверка ‹\p{Thai}› терпит неудачу), опережающая проверка терпит неудачу. Это приводит к тому, что вся попытка сопоставления признается неудачной, и механизм регулярных выражений вынужден начинать новую попытку со следующего символа.

Когда регулярное выражение достигает символа из алфавита Thai, проверка ‹\p{Thai}› проходит успешно. Вследствие этого и вся опережающая проверка ‹(?=\p{Thai})› признается совпавшей. Выходя из опережающей проверки, механизм регулярных выражений восстанавливает найденное ранее соответствие в прежнее состояние. В данном случае перед первым найденным символом алфавита Thai общее совпадение имело нулевую длину. Затем выполняется проверка ‹\p{N}›. Поскольку опе-

режающая проверка отменяет свое совпадение, подвыражение ‹\p{N}› сравнивается с тем же символом, с которым уже совпала проверка ‹\p{Thai}›. Если этот символ имеет свойство Number Юникода, проверка ‹\p{N}› совпадет. Поскольку подвыражение ‹\p{N}› находится за пределами какой-либо проверки соседних символов, оно поглощает символ и выражение обнаруживает тайскую цифру.

Проверки являются атомарными

Когда механизм регулярных выражений покидает группу проверки соседних символов, он отказывается от текста, совпавшего с проверкой. Поскольку текст совпадения отвергается, все позиции возврата, сохраненные операторами выбора или квантификаторами внутри проверки, также отвергаются. Это обеспечивает атомарность опережающих и ретроспективных проверок. В рецепте 2.14 во всех подробностях описывается атомарная группировка.

В большинстве ситуаций атомарная природа проверки остается невостребованной. Проверка соседних символов – это всего лишь попытка выяснить, совпадет ли регулярное выражение, находящееся внутри нее, или нет. Сколькими способами может произойти это совпадение, не имеет никакого значения, потому что проверка не поглощает ни одного символа из испытуемого текста.

Атомарность начинает обретать значение только при использовании сохраняющих групп внутри опережающей (и ретроспективной, если это допускает используемый диалект) проверки. Даже при том что опережающая проверка не поглощает текст, механизм регулярных выражений запоминает, какая часть текста совпала с любой из сохраняющих групп внутри опережающей проверки. Если опережающая проверка находится в конце регулярного выражения, в сохраняющих группах может оказаться совпавший с ними текст, не совпавший с самим регулярным выражением. Если опережающая проверка находится в середине регулярного выражения, может случиться так, что в сохраняющих группах окажутся перекрывающиеся участки испытуемого текста.

Единственная ситуация, когда атомарность проверки может повлиять на общее совпадение регулярного выражения, – это когда за пределами проверки используются обратные ссылки на сохраняющие группы, расположенные внутри проверки. Рассмотрим следующее регулярное выражение:

```
(?=(\d+))\w+\1
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

На первый взгляд, это регулярное выражение должно совпадать с текстом 123x12. Элемент ‹\d+› мог бы сохранить фрагмент 12 в первой сохраняющей группе, затем элемент ‹\w+› мог бы совпасть с фрагментом 3x, и наконец, ссылка ‹\1› могла бы совпасть с фрагментом 12.

Но этого никогда не произойдет. Механизм регулярных выражений входит в проверку и сохраняющую группу. Максимальный метасимвол ‹\d+› совпадает с фрагментом 123. Это совпадение сохраняется в первой сохраняющей группе. Затем механизм выходит из опережающей проверки и переустанавливает позицию поиска совпадений в начало строки, отвергая все позиции возврата, созданные максимальным квантификатором, но запоминает текст 123, сохраненный первой сохраняющей группой.

Теперь предпринимается попытка сопоставить максимальный элемент ‹\w+› с начала строки. Он поглощает весь текст 123x12. Сопоставление с обратной ссылкой ‹\1›, ссылающейся на текст 123, терпит неудачу в конце строки. Подвыражение ‹\w+› уступает один символ. Сопоставление с ‹\1› опять терпит неудачу. Подвыражение ‹\w+› продолжает уступать символы, пока у него не останется минимально возможный фрагмент 1. Сопоставление с обратной ссылкой ‹\1› в позиции, следующей за первым символом 1, также терпит неудачу.

Заключительный фрагмент текста 12 мог бы совпасть с обратной ссылкой ‹\1›, если бы механизм регулярных выражений мог вернуться в опережающую проверку и отказаться от совпадения 123 в пользу совпадения с 12, но он этого не делает.

Механизм регулярных выражений исчерпал все позиции возврата. Подвыражение ‹\w+› уступило все, что могло, а опережающая проверка, выполнившая подвыражение ‹\d+›, отбросила все позиции возврата. В результате попытка отыскать совпадение с регулярным выражением терпит неудачу.

Альтернатива ретроспективной проверке

```
<b>\K\w+(?=</b>)
```

Параметры: нечувствительность к регистру символов
Диалекты: PCRE 7.2, Perl 5.10

Perl 5.10, PCRE 7.2 и более поздние версии поддерживают альтернативный ретроспективным проверкам механизм в виде метасимвола ‹\K›. Когда механизм регулярных выражений встречает метасимвол ‹\K› в регулярном выражении, он *сохраняет* текст текущего совпадения. Сопоставление будет продолжено, как если бы метасимвол ‹\K› вообще отсутствовал в регулярном выражении. Но текст совпадения, найденного до встречи метасимвола ‹\K›, не будет включен в общий результат. Текст, совпавший с любыми сохраняющими группами до метасимвола ‹\K›, по-прежнему будет доступен для обратных ссылок после ‹\K›. Метасимвол ‹\K› оказывает влияние только на общий результат сопоставления.

Суть в том, что во многих ситуациях метасимвол ‹\K› можно использовать вместо положительной ретроспективной проверки. Выражение ‹before\Ktext› совпадет с фрагментом text, но только если ему предшествует фрагмент before, подобно выражению ‹(?<=before)text›. Преимущество

метасимвола ‹\K› перед положительной ретроспективной проверкой в диалектах Perl и PCRE состоит в том, что он позволяет использовать все возможности механизма регулярных выражений, тогда как ретроспективные проверки имеют некоторые ограничения, например недопустимость применения квантификаторов.

Главное отличие метасимвола ‹\K› от ретроспективной проверки в том, что при использовании ‹\K› сопоставление выполняется строго слева направо. То есть механизм регулярных выражений не заглядывает назад. Ретроспективные проверки заставляют его возвращаться. Эта разница становится существенной в ситуациях, когда часть регулярного выражения, следующая за метасимволом ‹\K› или за ретроспективной проверкой, может совпадать с тем же текстом, с которым совпала часть регулярного выражения, предшествующая метасимволу ‹\K› или находящаяся внутри ретроспективной проверки.

Регулярное выражение ‹(?<=a)a› найдет два совпадения в строке aaa. Первая попытка найти совпадение в начале строки потерпит неудачу, потому что механизм регулярных выражений ничего не обнаружит при попытке выполнить ретроспективную проверку. Попытка найти совпадение, начиная с позиции между первым и вторым символами a, увенчается успехом. Заглянув назад, механизм регулярных выражений обнаружит первый символ a в строке, удовлетворяющий условиям ретроспективной проверки. После этого второй элемент ‹a› в регулярном выражении совпадет со вторым символом a в строке. Третья попытка сопоставления будет предпринята в позиции между вторым и третьим символами a и также увенчается успехом. Заглянув назад, механизм регулярных выражений обнаружит второй символ a в строке, удовлетворяющий условиям ретроспективной проверки. Затем будет найдено совпадение с третьим символом a. Последняя попытка сопоставления будет предпринята в конце строки и потерпит неудачу. Заглянув назад, механизм регулярных выражений обнаружит третий символ a, удовлетворяющий условиям ретроспективной проверки. Но для второго элемента ‹a› в строке не останется символов.

Регулярное выражение ‹a\Ka› обнаружит только одно совпадение в строке. Первая же попытка сопоставления увенчается успехом. Первый элемент ‹a› в регулярном выражении совпадет с первым символом a в строке. Метасимвол ‹\K› исключит эту часть совпадения из общего возвращаемого результата, но никак не повлияет на сам процесс сопоставления. Второй элемент ‹a› в регулярном выражении затем совпадет со вторым символом a в строке, который и будет возвращен в виде общего совпадения. Вторая попытка сопоставления начнется между вторым и третьим символами a. Первый элемент ‹a› совпадет с третьим символом a. Метасимвол ‹\K› исключит его из общего результата, но механизм регулярных выражений продолжит процесс сопоставления как ни в чем не бывало. Однако для второго элемента ‹a› в строке не останется символов, поэтому попытка сопоставления потерпит неудачу.

Как видите, применение метасимвола ‹\K› не влияет на процесс сопоставления регулярного выражения. Регулярное выражение ‹a\Ka› будет обнаруживать те же совпадения, что и сохраняющая группа в выражении ‹a(a)›. Вы не сможете использовать ‹\K› для сопоставления с одной и той же частью строки более чем один раз, а ретроспективные проверки позволяют это. Например, чтобы найти совпадение с символом a, которому предшествует тайская цифра (то есть символ, одновременно являющийся символом алфавита Thai и цифрой), можно использовать выражение ‹(?<=\p{Thai})(?<=\p{Nd})a›. Если попытаться использовать выражение ‹\p{Thai}\K\p{Nd}\Ka›, оно обнаружит совпадение с фрагментом, состоящим из символа алфавита Thai, за которым следуют цифра и символ a, но вернет только символ a. И снова здесь наблюдается сходство с выражением ‹\p{Thai}\p{Nd}(a)›, совпадающим со всеми тремя символами и использующим сохраняющую группу для захвата части совпадения.

Решение без использования ретроспективной проверки

Все предыдущие замысловатые пояснения будут бесполезны, если вы пользуетесь Ruby 1.8 или JavaScript, потому что вы вообще не сможете воспользоваться ретроспективными проверками. Нет никакого способа решить эту проблему с помощью этих диалектов, но можно обойтись без ретроспективной проверки, применив сохраняющие группы. Это альтернативное решение также будет работать во всех других диалектах регулярных выражений.

```
(<b>)(\w+)(?=</b>)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Вместо ретроспективной проверки мы использовали сохраняющую группу для открывающего тега ‹<b>›. Интересующую нас часть регулярного выражения, ‹\w+›, мы также поместили в сохраняющую группу.

Если применить это регулярное выражение к тексту `My <b>cat</b> is furry`, общим совпадением для всего регулярного выражения будет фрагмент <b>cat. Первая сохраняющая группа захватит фрагмент <b>, а вторая – фрагмент cat.

Если требуется, чтобы совпадение содержало только слово cat (слово между тегами <b>), – например, чтобы выделить из текста только слово, – добиться этого можно простым сохранением не всего совпадения с регулярным выражением, а только текста, совпавшего со второй сохраняющей группой.

Если требуется выполнить поиск с заменой и заменить только слово между тегами, можно просто использовать обратную ссылку на первую сохраняющую группу, чтобы повторно вставить в текст открывающий тег в замещаемый текст. На самом деле, в данном случае сохраняющая группа вообще не нужна, так как открывающий тег всегда остается од-

ним и тем же. Но когда тег может изменяться, сохраняющая группа вставит именно тот тег, который совпал с ней. Подробнее этот прием рассматривается в рецепте 2.21.

Наконец, при желании можно имитировать ретроспективную проверку с помощью двух регулярных выражений. Первое выполняет поиск без ретроспективной проверки. Когда обнаруживается совпадение, нужно скопировать часть испытуемого текста, находящуюся перед совпадением, в новую строковую переменную. Второе регулярное выражение реализует проверку, которая выполнялась в ретроспективной проверке, при этом в конец выражения следует добавить привязку к концу строки (‹\z› или ‹$›). Якорный метасимвол обеспечит совпадение конца второго выражения с концом строки. Поскольку строка извлекается из текста в позиции, где совпало первое выражение, это обеспечит выполнение второй проверки непосредственно перед совпадением с первым выражением.

В JavaScript этот алгоритм можно было бы реализовать так:

```
var mainregexp = /\w+(?=<\/b>)/;
var lookbehind = /<b>$/;
if (match = mainregexp.exec("My <b>cat</b> is furry")) {
    // Найдено слово перед закрывающим тегом </b>
    var potentialmatch = match[0];
    var leftContext = match.input.substring(0, match.index);
    if (lookbehind.exec(leftContext)) {
        // Ретроспективная проверка совпала:
        // содержимое переменной potentialmatch находится между тегами <b>
    } else {
        // Ретроспективная проверка не совпала:
        // переменная potentialmatch содержит не то, что требуется
    }
} else {
    // Слово перед закрывающим тегом </b> не найдено
}
```

См. также

Рецепты 5.5, 5.6 и 7.10, где приводятся решения некоторых практических задач с применением проверок.

2.17. Совпадение с одной из двух альтернатив по условию

Задача

Создать регулярное выражение, совпадающее со списком слов one, two и three, разделенных запятыми. Эти слова присутствуют в списке не менее чем в одном экземпляре и могут следовать в произвольном порядке.

Решение

```
\b(?:(?:(one)|(two)|(three))(?:,|\b)){3,}(?(1)|(?!))(?(2)|(?!))(?(3)|(?!))
```

Параметры: нет
Диалекты: .NET, PCRE, Perl, Python

Диалекты Java, JavaScript и Ruby не поддерживают условные операторы. При программировании на этих языках (или любых других) можно использовать регулярные выражения без условных операторов, написав дополнительный программный код, проверяя совпадения с каждой из трех сохраняющих групп.

```
\b(?:(?:(one)|(two)|(three))(?:,|\b)){3,}
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Диалекты .NET, PCRE, Perl и Python поддерживают *условные операторы* при использовании нумерованных сохраняющих групп. Выражение ‹(?(1)*then*|*else*)› – это условный оператор, проверяющий совпадение с первой сохраняющей группой. Если группа участвовала в совпадении, механизм регулярных выражений попытается найти совпадение ‹*then*›. Если сохраняющая группа до сих пор не участвовала в совпадении, выполняется попытка сопоставить часть ‹*else*›.

Круглые скобки, знак вопроса и вертикальная черта – все эти символы составляют синтаксис условного оператора. В данном случае их значение отличается от обычного. В частях ‹*then*› и ‹*else*› допускается использовать любые регулярные выражения. Единственное ограничение состоит в том, что при необходимости использовать оператор выбора в какой-либо из частей альтернативы следует заключать в группу. В условном операторе допускается использовать только один символ вертикальной черты.

При желании можно опустить любую из частей ‹*then*› или ‹*else*›. Пустое регулярное выражение всегда имеет совпадение нулевой длины. В решении этой задачи используются три условных оператора, в которых опущена часть ‹*then*›. Если сохраняющая группа участвовала в сопоставлении, считается, что условие выполняется.

Пустая отрицательная опережающая проверка, ‹(?!)›, заполняет часть ‹*else*›. Поскольку пустое регулярное выражение всегда совпадает, отрицательная опережающая проверка, содержащая пустое регулярное выражение, всегда будет терпеть неудачу. То есть условие ‹(?(1)|(?!))› никогда не будет выполняться, если первая сохраняющая группа не совпала с чем-нибудь.

Поместив каждую из трех обязательных альтернатив в отдельную сохраняющую группу, можно использовать три условных оператора в кон-

це регулярного выражения, чтобы проверить наличие совпадений для каждой сохраняющей группы. Если хотя бы с одним из слов не будет найдено совпадение, условная ссылка на соответствующую сохраняющую группу выполнит часть ‹*else*›, которая вызовет неудачу сопоставления из-за наличия пустой отрицательной опережающей проверки в этой части. То есть регулярное выражение будет терпеть неудачу, если в испытуемом тексте отсутствует хотя бы одно искомое слово.

Чтобы порядок следования слов и число их вхождений в испытуемом тексте не оказывали влияния на результат, мы заключили оператор выбора в группу и добавили квантификатор для этой группы. Так как по условиям задачи требуется определить присутствие всех трех слов, сопоставление с группой необходимо повторить не менее трех раз.

Диалекты .NET, Python и PCRE 6.7 позволяют ссылаться на именованные сохраняющие группы в условных операторах. Выражение ‹(?(*name*)*then*|*else*)› проверяет, участвовала ли сохраняющая группа с именем *name* в сопоставлении к настоящему моменту. Версии Perl 5.10 и выше также поддерживают именованные условные операторы. Но Perl требует заключать имя в угловые скобки или кавычки, например: ‹(?(<*name*>)*then*|*else*)› или ‹(?('*name*')*then*|*else*)›. Версии PCRE 7.0 и выше тоже поддерживают синтаксис Perl именованных условных операторов, но при этом также поддерживают синтаксис .NET и Python.

Чтобы лучше понять, как работают условные операторы, рассмотрим регулярное выражение ‹(a)?b(?(1)c|d)›. Это, по сути, более сложная форма записи ‹abc|bd)›.

Если испытуемый текст начинается с символа a, он сохраняется в первой сохраняющей группе. В противном случае первая сохраняющая группа вообще не участвует в сопоставлении. Знак вопроса, находящийся за пределами сохраняющей группы, имеет важное значение, так как он делает всю группу необязательной. Если символ a отсутствует, группа повторяется ноль раз и никогда не получит возможность сохранить что-либо. Она не может сохранить строку нулевой длины.

Если бы использовалось подвыражение ‹(a?)›, группа всегда участвовала бы в сопоставлении. Так как в этом подвыражении после группы нет квантификатора, она повторялась бы точно один раз. Группа сохранила бы символ a либо не сохранила ничего.

Независимо от наличия совпадения с ‹a›, далее идет следующая лексема ‹b›. За ней следует проверка условия. Если сохраняющая группа участвовала в сопоставлении, даже если она сохранит строку нулевой длины (что здесь невозможно), будет предпринята попытка найти соответствие для ‹c›. В противном случае будет произведена попытка сопоставления с ‹d›.

Говоря простым языком, выражение ‹(a)?b(?(1)c|d)› либо совпадает с фрагментом ab, за которым следует c, либо совпадает с b, за которым следует d.

В диалектах .NET, PCRE и Perl, но не в диалекте Python, условные операторы могут также выполнять проверку соседних символов. Выражение ‹(?(?=*if*)*then*|*else*)› сначала выполняет ‹(?=*if*)› как обычную опережающую проверку. Как это работает, описывается в рецепте 2.16. Если проверка увенчалась успехом, выполняется попытка сопоставления с частью ‹*then*›, в противном случае – с частью ‹*else*›. Поскольку проверка ничего не поглощает из испытуемого текста, обе части ‹*then*› и ‹*else*› применяются к одной и той же позиции в испытуемом тексте, где производилась проверка условия ‹*if*›, независимо от результата проверки.

В условном операторе вместо опережающей проверки можно использовать ретроспективную проверку. Можно также использовать отрицательные проверки, хотя мы не рекомендуем делать это, так как можно только все запутать, перевернув значения подвыражений «then» и «else».

Выражение с условием, использующим проверку, может быть записано без применения условного оператора, например: ‹(?=*if*)*then*|(?!*if*)*else*›. Если положительная опережающая проверка дает положительный результат, выполняется попытка сопоставления части ‹*then*›. В противном случае выполняется попытка сопоставить альтернативный вариант. Затем то же самое делает отрицательная опережающая проверка. Отрицательная опережающая проверка завершается успехом, когда подвыражение ‹*if*› не находит соответствия, что происходит гарантированно, так как проверка ‹(?=*if*)› уже потерпела неудачу. В результате предпринимается попытка сопоставить подвыражение ‹*else*›. Размещение опережающей проверки в условном операторе экономит время, так как в этом случае условие ‹*if*› проверяется всего один раз.

См. также

Условные операторы по своей сути являются комбинациями проверок (рецепт 2.16) и операторов выбора (рецепт 2.8) внутри групп (рецепт 2.9).

Разделы «Пропуск некорректных идентификаторов ISBN» (рецепт 4.13) и «Использование условных конструкций» (рецепт 5.7), где показаны приемы решения некоторых практических задач с использованием условных операторов.

2.18. Добавление комментариев в регулярные выражения

Задача

Выражение ‹\d{4}-\d{2}-\d{2}› совпадает с датами в формате уууу-mm-dd, но не выполняет проверку чисел на корректность. Такое простое регулярное выражение вполне может использоваться на практике, когда заранее известно, что текст не может содержать некорректные даты.

Необходимо добавить комментарии в это регулярное выражение, чтобы описать, что делают его части.

Решение

```
\d{4} # Год
-     # Разделитель
\d{2} # Месяц
-     # Разделитель
\d{2} # День
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Обсуждение

Режим свободного форматирования

Регулярные выражения быстро становятся очень сложными и трудными для понимания. Так же как и при работе с исходными текстами программ, желательно комментировать все регулярные выражения, за исключением самых тривиальных.

Все диалекты регулярных выражений, рассматриваемые в этой книге, за исключением JavaScript, предлагают альтернативный синтаксис оформления регулярных выражений, который существенно упрощает добавление комментариев в регулярные выражения. На использование этого синтаксиса можно перейти, включив параметр *свободного форматирования*. В различных языках программирования этот параметр носит разные имена.

В .NET необходимо включить параметр `RegexOptions.IgnorePatternWhitespace`. В Java – передать флаг `Pattern.COMMENTS`. В Python этот флаг называется `re.VERBOSE`, в PHP, Perl и Ruby для этих целей используется флаг `/x`.

Хотя стандартный диалект JavaScript не поддерживает режим свободного форматирования регулярных выражений, библиотека XRegExp предоставляет такую возможность. Для этого достаточно просто добавить `'x'` в список флагов, передаваемых конструктору `XRegExp()` во втором параметре.

Включение режима свободного форматирования влечет за собой появление двух эффектов. В этом режиме в число метасимволов включается символ решетки (#), имеющий специальное назначение за пределами символьных классов. С символа решетки начинаются комментарии, которые продолжаются до конца строки или до конца регулярного выражения (в зависимости от того, какое условие выполнится первым). Символ решетки и все, что следует за ним, просто игнорируются механизмом регулярных выражений. Чтобы оформить поиск совпадения с символом решетки, его следует поместить в символьный класс ‹`[#]`› или экранировать ‹`\#`›.

Другой эффект состоит в том, что пробельные символы, в число которых входят пробелы, символы табуляции и символы перевода строки, также игнорируются за пределами символьных классов. Чтобы оформить поиск совпадения с символом пробела, его следует поместить в символьный класс ‹[ ]› или экранировать ‹\ ›. Для обеспечения большей удобочитаемости можно использовать шестнадцатеричную экранированную последовательность ‹\x20›, или кодовый пункт Юникода ‹\u0020›, или ‹\x{0020}›. Чтобы оформить поиск совпадения с символом табуляции, следует использовать метасимвол ‹\t›. Для поиска конца строки следует использовать комбинацию ‹\r\n› (Windows) или ‹\n› (UNIX/Linux/OS X).

Режим свободного форматирования не влияет на порядок оформления символьных классов. Символьный класс – это единый элемент. Любые пробельные символы или символы решетки внутри символьных классов рассматриваются как литералы, добавленные в символьный класс. Нет никакой возможности разбить символьный класс на отдельные части, чтобы прокомментировать их.

Символьные классы Java, поддерживающие свободное форматирование

Регулярные выражения едва ли заслужили бы столь высокую репутацию, если бы один диалект был несовместим с другими. В данном случае Java – исключение из правил.

В диалекте Java символьные классы не являются едиными и неделимыми элементами. При включении режима свободного форматирования диалект Java начинает игнорировать пробельные символы в символьных классах, а символы решетки в символьных классах играют роль начала комментариев. Это означает потерю возможности использовать ‹[•]› и ‹[#]› для поиска соответствий с литералами этих символов. Вместо них придется использовать экранированные последовательности ‹\u0020› и ‹\#›.

Варианты

```
(?#Year)\d{4}(?#Separator)-(?#Month)\d{2}-(?#Day)\d{2}
```

Параметры: нет

Диалекты: .NET, XRegExp, PCRE, Perl, Python, Ruby

Если по каким-то причинам невозможно или нежелательно использовать режим свободного форматирования, можно добавлять комментарии в форме ‹(?#comment)›. Все символы между ‹(?#› и ‹)› будут игнорироваться.

К сожалению, диалект JavaScript, который единственный из рассматриваемых в этой книге не поддерживает режим свободного форматирования, также не поддерживает и альтернативный синтаксис оформления комментариев. Библиотека XRegExp, добавляющая поддержку свободного форматирования регулярных выражений в JavaScript, также

добавляет и поддержку синтаксиса комментариев. Хотя диалект Java поддерживает комментарии в режиме свободного форматирования регулярных выражений, он не поддерживает синтаксис ‹(?#comment)›.

```
(?x)\d{4} # Год
-         # Разделитель
\d{2}     # Месяц
-         # Разделитель
\d{2}     # День
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Если нет возможности включить режим свободного форматирования за пределами регулярного выражения, можно в самое начало регулярного выражения добавить модификатор ‹(?x)›. Обратите внимание, что перед модификатором ‹(?x)› не должно быть пробелов. Режим свободного форматирования включается только начиная с позиции этого модификатора – любые пробелы перед ним имеют значение.

Подробнее о модификаторах рассказывается в рецепте 2.1 в подразделе «Поиск без учета регистра символов».

2.19. Вставка текстового литерала в замещающий текст

Задача

Для любого регулярного выражения, выполняющего поиск с заменой совпадения, подготовить замещающий текст со следующими символами: $%*$1\1.

Решение

```
$%\*$$1\1
```

Диалекты замещающего текста: .NET, JavaScript

```
\$%\\*\$1\\1
```

Диалект замещающего текста: Java

```
$%\*\$1\\1
```

Диалект замещающего текста: PHP

```
\$%\*\$1\\1
```

Диалект замещающего текста: Perl

```
$%\*$1\\1
```

Диалекты замещающего текста: Python, Ruby

Обсуждение

Когда и как экранировать символы в замещающем тексте

Данный рецепт демонстрирует разные правила экранирования, используемые в разных диалектах замещающего текста. Единственные два символа, которые может потребоваться экранировать в замещающем тексте, – это знак доллара и символ обратного слэша. Символами экранирования также являются знак доллара и символ обратного слэша. Знак процента и звездочка в этом примере всегда интерпретируются как обычные литералы, однако предшествующий символ обратного слэша может интерпретироваться как экранирующий символ, а не как литерал. «$1» и/или «\1» являются обратными ссылками на сохраняющую группу. В рецепте 2.21 описывается, в каком диалекте какой синтаксис используется для оформления обратных ссылок.

Тот факт, что эта задача имеет пять решений для семи диалектов замещающего текста, демонстрирует отсутствие стандартного синтаксиса для замещающего текста.

.NET и JavaScript

Диалекты .NET и JavaScript всегда интерпретируют символ обратного слэша как литерал. Его не требуется экранировать другим символом обратного слэша, в противном случае замещающий текст будет содержать два символа обратного слэша.

Одиночный знак доллара рассматривается как литерал. Знак доллара требуется экранировать, только если за ним следуют цифра, амперсанд, обратный апостроф, прямая кавычка, символ подчеркивания, знак плюс или другой знак доллара. Чтобы экранировать знак доллара, перед ним нужно поместить еще один знак доллара. Все знаки доллара можно продублировать, если вам кажется, что это сделает замещающий текст более удобочитаемым. Следующее решение совершенно идентично тому, что приводится выше.

```
$$%\*$$1\1
```

Диалект замещающего текста: .NET, JavaScript

Кроме того, в диалектах .NET и XRegExp необходимо экранировать знак доллара, предшествующий открывающей фигурной скобке. В диалектах .NET и XRegExp последовательность «${group}» интерпретируется как именованная обратная ссылка. Диалект JavaScript не поддерживает именованные обратные ссылки без использования библиотеки XRegExp.

Java

В диалекте Java для экранирования обратного слэша и знака доллара в замещающем тексте используется символ обратного слэша. Все литералы обратного слэша и литералы знака доллара должны экранироваться

символом обратного слэша. Если этого не сделать, будет возбуждено исключение.

PHP

PHP требует, чтобы все цифры и знаки доллара, предшествующие цифре или открывающей фигурной скобке, экранировались символом обратного слэша.

Обратный слэш экранирует другой символ обратного слэша. Так, чтобы вставить в замещающий текст два символа слэша, их следует записать как «\\\\». Все остальные символы обратного слэша интерпретируются как литералы.

Perl

Диалект Perl несколько отличается от других диалектов замещающего текста: в действительности в нем нет диалекта замещающего текста. Тогда как в других языках программирования в процедурах поиска с заменой присутствует специальная логика, выполняющая подстановку для таких конструкций, как «$1», в языке Perl это обычная интерпретация переменной. В замещающем тексте необходимо экранировать символом обратного слэша все литералы знака доллара, как если бы они присутствовали в строке, ограниченной кавычками.

Единственное исключение состоит в том, что в языке Perl поддерживается синтаксис «\1» оформления обратных ссылок. Поэтому необходимо экранировать символ обратного слэша, за которым следует цифра, если обратный слэш должен интерпретироваться как литерал. Обратный слэш, за которым следует знак доллара, также должен экранироваться, чтобы предотвратить экранирование знака доллара.

Обратный слэш экранирует другой символ обратного слэша. Так, чтобы вставить в замещающий текст два символа слэша, их следует записать как «\\\\». Все остальные символы обратного слэша интерпретируются как литералы.

Python и Ruby

В диалектах Python и Ruby знак доллара не имеет специального значения в замещающем тексте. Символы обратного слэша должны экранироваться другими обратными слэшами, если за ними следует символ, придающий обратному слэшу специальное значение.

В диалекте Python конструкции с «\1» по «\9» и «\g<» образуют обратные ссылки. Эти символы обратного слэша необходимо экранировать.

В диалекте Ruby необходимо экранировать символ обратного слэша, за которым следует цифра, амперсанд, обратный апостроф, прямая кавычка или знак плюс.

Кроме того, в обоих языках обратный слэш экранирует другой символ обратного слэша. Так, чтобы вставить в замещающий текст два символа

слэша, их следует записать как «\\\\». Все остальные символы обратного слэша интерпретируются как литералы.

Дополнительные правила экранирования для строковых литералов

Не забывайте, что в этой главе мы имеем дело только с регулярными выражениями и замещающим текстом как таковыми. Языки программирования и строковые литералы будут рассматриваться в следующей главе.

Любой замещающий текст из продемонстрированных выше сможет применяться, только когда функции `replace()` передается фактическая строковая переменная, хранящая этот текст. Другими словами, если в приложении имеется поле ввода, куда пользователь вводит замещающий текст, эти решения демонстрируют, что должен вводить пользователь, чтобы функция поиска с заменой работала, как ожидается. Если проверить команды поиска с заменой в RegexBuddy или в другой программе проверки регулярных выражений, этот рецепт продемонстрирует ожидаемые результаты.

Но те же замещающие тексты не будут работать, если вставить их непосредственно в исходный текст программы и окружить кавычками. Строковые литералы в языках программирования определяют собственные правила экранирования, и этим правилам необходимо следовать, применяя их поверх правил экранирования, определяемых диалектом замещающего текста. На практике в конечном итоге может получиться частокол из символов обратного слэша.

См. также

Рецепт 3.14, где демонстрируется, как добавлять операции поиска с заменой в исходный программный код.

2.20. Вставка совпадения с регулярным выражением в замещающий текст

Задача

Выполнить поиск с заменой, в процессе которого все адреса URL будут преобразованы в ссылки HTML, указывающие на эти адреса, и использовать обнаруженные адреса URL как замещающий текст. Для данного упражнения примем, что адреса URL начинаются с последовательности «http:», за которой следуют любые непробельные символы. Например, текст `Please visit http://www.regexcookbook.com` должен превратиться в текст `Please visit <a href="http://www.regexcookbook.com">http://www.regexcookbook.com</a>`.

Решение

Регулярное выражение

```
http:\S+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
<a href="$&">$&</a>
```

Диалекты замещающего текста: .NET, JavaScript, Perl

```
<a href="$0">$0</a>
```

Диалекты замещающего текста: .NET, Java, XRegExp, PHP

```
<a href="\0">\0</a>
```

Диалекты замещающего текста: PHP, Ruby

```
<a href="\&">\&</a>
```

Диалект замещающего текста: Ruby

```
<a href="\g<0>">\g<0></a>
```

Диалект замещающего текста: Python

Обсуждение

Вставка всего совпадения с регулярным выражением обратно в замещающий текст обеспечивает простой способ вставки нового текста перед, после или вокруг совпавшего текста или даже между несколькими копиями совпавшего текста. Чтобы использовать совпадение со всем регулярным выражением, не потребуется добавлять какие-либо сохраняющие группы в регулярное выражение, за исключением диалекта Python.

В языке Perl комбинация «$&» фактически является переменной. В случае успеха в этой переменной сохраняется общее совпадение с регулярным выражением. Однако использование конструкции «$&» отрицательно сказывается на производительности любых регулярных выражений в Perl, поэтому может оказаться предпочтительнее заключить все регулярное выражение в сохраняющую группу и использовать обратную ссылку на эту группу.

Диалекты .NET и JavaScript поддерживают синтаксис «$&» для вставки совпадения с регулярным выражением в замещающий текст. В диалекте Ruby в аналогичной конструкции вместо знака доллара используется символ обратного слэша, поэтому общее совпадение может быть вставлено с помощью последовательности «\&».

В диалектах Java, PHP и Python отсутствует специальный элемент для вставки совпадения со всем регулярным выражением, но они позволяют

вставлять в замещающий текст совпадение с сохраняющей группой, как описывается в следующем разделе. Все регулярное выражение в них рассматривается как неявная сохраняющая группа с номером 0. Чтобы сослаться на нулевую сохраняющую группу в языке Python, необходимо использовать синтаксис именованных обратных ссылок. Python не поддерживает синтаксис «\0».

Диалекты .NET, XRegExp и Ruby также поддерживают синтаксис обращения к нулевой сохраняющей группе, но в них совершенно неважно, какой синтаксис используется. Результат будет тем же самым.

См. также

Различные диалекты замещающего текста описываются в разделе «Поиск с заменой с помощью регулярных выражений» в главе 1.

Рецепт 3.15, где рассказывается, как использовать текст замены в исходном программном коде.

2.21. Вставка части совпадения с регулярным выражением в замещающий текст

Задача

Найти совпадение с любой непрерывной последовательностью из 10 цифр, например 1234567890. Преобразовать эту последовательность в формат представления телефонных номеров, например (123) 456-7890.

Решение

Регулярное выражение

```
\b(\d{3})(\d{3})(\d{4})\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
($1)•$2-$3
```

Диалекты замещающего текста: .NET, Java, JavaScript, PHP, Perl

```
(${1})•${2}-${3}
```

Диалекты замещающего текста: .NET, PHP, Perl

```
(\1)•\2-\3
```

Диалекты замещающего текста: PHP, Python, Ruby

Обсуждение

Замещение с использованием сохраняющих групп

В рецепте 2.10 описывалось, как использовать сохраняющие группы в регулярном выражении, чтобы получить более одного совпадения с одним и тем же текстом. Текст, совпавший с каждой из сохраняющих групп, также будет доступен после каждого успешного совпадения. Текст из некоторых или всех сохраняющих групп можно вставлять в замещающий текст в любом порядке и даже несколько раз.

Некоторые диалекты, такие как Python и Ruby, используют одинаковый синтаксис «\1» обратных ссылок как в регулярных выражениях, так и в замещающем тексте. Другие диалекты используют синтаксис Perl «$1», где вместо обратного слэша применяется знак доллара. PHP поддерживает оба варианта.

В языке Perl ссылки «$1» и с иными цифрами фактически являются переменными, значения которых устанавливаются после каждого успешного совпадения регулярного выражения. Их можно использовать в любом месте программного кода, пока не будет предпринята новая попытка сопоставления регулярного выражения. Диалекты .NET, Java, JavaScript и PHP поддерживают синтаксис «$1» только в замещающем тексте. Для доступа к содержимому сохраняющих групп в программном коде на этих языках программирования предоставляются иные способы. Более подробно об этом рассказывается в главе 3.

$10 и выше

Все диалекты регулярных выражений, рассматриваемые в этой книге, поддерживают до 99 сохраняющих групп в регулярных выражениях. В замещающем тексте может возникать неоднозначность интерпретации обратных ссылок для «$10» или «\10» и выше. Они могут интерпретироваться и как обратная ссылка на группу с номером 10, и как обратная ссылка на первую группу, за которой следует литерал «0».

Диалекты .NET, XRegExp, PHP и Perl позволяют помещать номер сохраняющей группы в фигурные кавычки, устраняя неоднозначность. Последовательность «${10}» всегда обозначает обратную ссылку на сохраняющую группу с номером 10, а последовательность «${1}0» всегда обозначает обратную ссылку на первую сохраняющую группу, за которой следует литерал «0».

Диалекты Java и JavaScript пытаются правильно интерпретировать последовательность «$10». Если в регулярном выражении существует сохраняющая группа с указанным двузначным номером, такая последовательность будет интерпретироваться как обратная ссылка на сохраняющую группу. Если число сохраняющих групп меньше, только первая цифра будет использоваться как ссылка на группу, а ноль будет интерпретироваться как литерал. То есть последовательность «$23»

будет означать ссылку на сохраняющую группу с номером 23, только если таковая существует в регулярном выражении. В противном случае она будет обозначать ссылку на вторую сохраняющую группу, за которой следует литерал «3».

Диалекты .NET, XRegExp, PHP, Perl, Python и Ruby всегда интерпретируют последовательность «$10» и «\10» как ссылку на сохраняющую группу с номером 10 независимо от ее существования. Если такой группы не существует, вступают в действия правила, применяемые к ссылкам на несуществующие группы.

Ссылки на несуществующие группы

Регулярное выражение, представленное в решении для этого рецепта, содержит три сохраняющих группы. Если в замещающем тексте указать последовательность «$4» или «\4», она будет интерпретироваться как ссылка на несуществующую сохраняющую группу. Это вызовет три варианта действий.

Java, XRegExp и Python завопят о нарушении, возбудив исключение или вернув сообщение об ошибке. В этих диалектах не следует использовать ошибочные обратные ссылки. (На самом деле, ошибочные обратные ссылки не следует использовать ни в одном из диалектов.) Если нужно, чтобы последовательность «$4» или «\4» интерпретировалась буквально, необходимо экранировать знак доллара или символ обратного слэша. Подробнее об этом рассказывается в рецепте 2.19.

PHP, Perl и Ruby подставляют в замещающий текст все ссылки, включая и те, что указывают на несуществующие сохраняющие группы. Несуществующие группы ничего не сохраняют, и потому ссылки, указывающие на такие группы, замещаются пустым текстом.

Наконец, .NET и JavaScript (без XRegExp) интерпретируют обратные ссылки на несуществующие сохраняющие группы как литеральный текст.

Все диалекты выполняют замещение ссылок на сохраняющие группы, существующие в регулярном выражении, но не сохранившие ничего. Такие ссылки замещаются пустым текстом.

Решение с использованием именованных сохранений

Регулярное выражение

```
\b(?<area>\d{3})(?<exchange>\d{3})(?<number>\d{4})\b
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?'area'\d{3})(?'exchange'\d{3})(?'number'\d{4})\b
```

Параметры: нет

Диалекты: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?P<area>\d{3})(?P<exchange>\d{3})(?P<number>\d{4})\b
```

Параметры: нет
Диалекты: PCRE, Perl 5.10, Python

Замещающий текст

```
(${area})·${exchange}-${number}
```

Диалекты замещающего текста: .NET, Java 7, XRegExp

```
(\g<area>)·\g<exchange>-\g<number>
```

Диалект замещающего текста: Python

```
(\k<area>)·\k<exchange>-\k<number>
```

Диалект замещающего текста: Ruby 1.9

```
(\k'area')·\k'exchange'-\k'number'
```

Диалект замещающего текста: Ruby 1.9

```
($+{area})·$+{exchange}-$+{number}
```

Диалект замещающего текста: Perl 5.10

```
($1)·$2-$3
```

Диалект замещающего текста: PHP

Диалекты, поддерживающие именованные сохранения

Диалекты .NET, Java 7, Python и Ruby 1.9 позволяют использовать в замещающем тексте именованные обратные ссылки, если в регулярном выражении используются именованные сохраняющие группы. Синтаксис именованных обратных ссылок для работы с именованными сохраняющими группами в тексте замены отличается от синтаксиса, используемого в регулярных выражениях.

В диалекте замещающего текста Ruby используется тот же синтаксис обратных ссылок, что и в регулярных выражениях. Для именованных групп в Ruby 1.9 синтаксис имеет вид: «\k<group>» или «\k'group'». Выбор между угловыми скобками и апострофами – это вопрос личных предпочтений.

В версиях Perl 5.10 и выше фрагменты, совпавшие с именованными сохраняющими группами, запоминаются в хеше %+, благодаря чему их можно извлекать по именам сохраняющих групп обращением к $+{name}. Perl выполняет интерполяцию переменных в замещающем тексте, поэтому текст «$+{name}» будет интерпретироваться как именованная обратная ссылка в тексте замены.

PHP (при использовании библиотеки PCRE) поддерживает именованные сохраняющие группы в регулярных выражениях, но не в замещающем тексте. Чтобы обратиться к именованным группам, присутствующим в регулярном выражении, в замещающем тексте можно использовать нумерованные обратные ссылки. Диалект PCRE присваивает номера именованным и нумерованным группам в направлении слева направо.

Диалекты .NET, Java 7, XRegExp, Python и Ruby 1.9 также позволяют использовать нумерованные ссылки на именованные группы. Однако для именованных групп в диалекте .NET используется иная схема нумерации, как описывается в рецепте 2.11. Смешивание именованных и нумерованных групп в диалектах .NET, Java 7, XRegExp, Python и Ruby не рекомендуется. Желательно, чтобы либо все группы были именованными, либо все – нумерованными. Для обращения к именованным группам всегда следует использовать именованные обратные ссылки.

См. также

Рецепт 2.9, где рассказывается о сохраняющих группах, к которым можно обращаться с помощью обратных ссылок.

Рецепт 2.11, где рассказывается об именованных сохраняющих группах. Возможность именования групп в регулярном выражении упрощает чтение и сопровождение регулярного выражения.

Различные диалекты замещающего текста описываются в разделе «Поиск с заменой с помощью регулярных выражений» в главе 1.

Рецепт 2.10, где демонстрируется, как использовать обратные ссылки в самом регулярном выражении. В тексте замены используется иной синтаксис определения обратных ссылок.

Рецепт 3.15, где рассказывается, как использовать текст замены в исходном программном коде.

2.22. Вставка контекста совпадения в замещающий текст

Задача

Создать замещающий текст, который заместит совпадение с регулярным выражением текстом, расположенным перед совпадением, всем испытуемым текстом и остатком испытуемого текста, расположенным после совпадения. Например, если в тексте `BeforeMatchAfter` был найден фрагмент `Match`, его следует заменить текстом `BeforeBeforeMatchAfterAfter`, в результате должен получиться текст `BeforeBeforeBeforeMatchAfterAfterAfter`.

Решение

```
$`$_$`
```

Диалекты замещающего текста: .NET, Perl

```
\`\`\&\`\`
```

Диалект замещающего текста: Ruby

```
$`$`$&$`$`
```

Диалект замещающего текста: JavaScript

Обсуждение

Под термином *контекст* подразумевается испытуемый текст, к которому применялось регулярное выражение. Контекст делится на три части: испытуемый текст перед совпадением с регулярным выражением, испытуемый текст после совпадения и весь испытуемый текст. Текст, расположенный перед совпадением, иногда называют *левым контекстом*, а текст, расположенный после совпадения, соответственно *правым контекстом*. Весь испытуемый текст образуется левым контекстом, совпадением и правым контекстом.

Диалекты .NET и Perl поддерживают конструкции «$`», «$'» и «$_», которые вставляют все три составляющие контекста в замещающий текст. Фактически в языке Perl эти конструкции являются переменными, значения которых устанавливаются после успешного совпадения с регулярным выражением и которые доступны в любом месте в программном коде, пока не будет выполнена новая попытка сопоставления. Комбинация «$`» обозначает левый контекст. Символ обратного апострофа вводится с клавиатуры в раскладке U.S. нажатием на клавишу, расположенную левее клавиши с цифрой 1 в левом верхнем углу клавиатуры. Комбинация «$'» обозначает правый контекст. В качестве прямой кавычки здесь используется обычный апостроф. На клавиатуре в раскладке U.S. клавиша с этим символом находится между клавишей с точкой с запятой и клавишей Enter. Комбинация «$_» обозначает весь испытуемый текст. Подобно диалектам .NET и Perl, диалект JavaScript использует комбинации «$`» и «$'» для обозначения левого и правого контекстов. Однако в JavaScript отсутствует элемент, соответствующий всему испытуемому тексту. При работе с этим диалектом можно восстановить весь испытуемый текст, вставив текст совпадения между левым и правым контекстами с помощью конструкции «$&».

В диалекте Ruby доступ к левому и правому контекстам поддерживается конструкциями «\`» и «\'», а текст совпадения с регулярным выражением вставляется конструкцией «\&». Как и в JavaScript, здесь нет элемента, с помощью которого можно было бы вставить весь испытуемый текст.

См. также

Различные диалекты замещающего текста описываются в разделе «Поиск с заменой с помощью регулярных выражений» в главе 1.

Рецепт 3.15, где рассказывается, как использовать текст замены в исходном программном коде.

3

Программирование с применением регулярных выражений

Языки программирования и диалекты регулярных выражений

В этой главе описывается, как реализовать регулярные выражения в различных языках программирования. Приведенные рецепты предполагают, что в вашем распоряжении уже имеется готовое регулярное выражение; в этом вы можете опираться на предыдущую главу. Теперь перед вами стоит задача поместить регулярное выражение в исходный текст программы и заставить его работать.

В этой главе мы приложили максимум усилий, чтобы объяснить, как и почему каждый фрагмент программного кода работает тем или иным способом. Высокий уровень детализации может сделать чтение этой главы от начала до конца достаточно утомительным занятием. Если вы читаете книгу «Регулярные выражения. Сборник рецептов» в первый раз, мы рекомендуем бегло просмотреть эту главу, чтобы получить общее представление о том, что может потребоваться. Позднее, когда появится необходимость реализовать какое-либо из регулярных выражений, рассматриваемых в последующих главах, можно будет вернуться сюда и подробнее узнать, как интегрировать регулярные выражения в программу, написанную на том или ином языке программирования.

В главах с 4 по 9 демонстрируются регулярные выражения, предназначенные для решения практических задач. Основное внимание в этих главах уделяется самим регулярным выражениям, и во многих рецептах программный код вообще не демонстрируется. Чтобы заставить какое-то регулярное выражение, приведенное в следующих главах, работать, просто нужно включить его в один из фрагментов программного кода, которые демонстрируются в данной главе.

Поскольку последующие главы основное внимание уделяют именно регулярным выражениям, они представляют решения для определенных диалектов регулярных выражений, а не для языков программирования. Между диалектами регулярных выражений и языками программирования нет точного соответствия. Языки сценариев обычно используют собственный встроенный диалект регулярных выражений, другие языки программирования опираются на библиотеки, реализующие поддержку регулярных выражений. Некоторые библиотеки могут использоваться в нескольких языках программирования, а в некоторых языках программирования имеется возможность использовать разные библиотеки. В разделе «Множество диалектов регулярных выражений» (глава 1) описываются все диалекты регулярных выражений, рассматриваемые в этой книге. В разделе «Множество диалектов определения замещающего текста» (глава 1) перечислены диалекты определения замещающего текста, используемого в операциях поиска с заменой, с применением регулярных выражений. Все языки программирования, описываемые в этой главе, используют один из этих диалектов.

Языки, рассматриваемые в этой главе

В этой главе рассматриваются восемь языков программирования. Каждый рецепт имеет отдельные решения для всех языков, а во многих рецептах также даются отдельные пояснения для всех восьми языков. Если применяемый прием относится сразу к нескольким языкам, мы повторим его в пояснениях для каждого из этих языков. Вы без всякого ущерба для себя можете пропускать пояснения для языков программирования, которые вас не интересуют.

C#

Язык программирования C# используется платформой Microsoft .NET Framework. Классы `System.Text.RegularExpressions` используют диалект регулярных выражений и диалект замещающего текста «.NET». В этой книге охватываются версии C# с 1.0 по 4.0 или Visual Studio с версии 2002 по 2010.

VB.NET

Под названиями VB.NET и Visual Basic.NET, используемыми в этой книге, подразумевается Visual Basic 2002 и более поздние версии, чтобы обозначить отличие этих версий от Visual Basic 6 и более ранних. В настоящее время Visual Basic использует платформу Microsoft .NET Framework. Классы `System.Text.RegularExpressions` используют диалект регулярных выражений и диалект замещающего текста «.NET». В этой книге охватываются версии Visual Basic с 2002 по 2010.

Java

Версия Java 4 стала первой версией, обеспечившей встроенную поддержку регулярных выражений в виде пакета `java.util.regex`. Пакет `java.util.regex` использует диалект регулярных выражений и диалект

замещающего текста «Java». В этой книге охватываются версии Java 4, 5, 6 и 7.

JavaScript

Этот диалект регулярных выражений используется в языке программирования, который наиболее широко известен под названием JavaScript. Все современные веб-браузеры реализуют поддержку этого языка программирования: Internet Explorer (начиная с версии 5.5), Firefox, Opera, Safari и Chrome. Многие приложения также используют JavaScript в качестве языка сценариев.

Строго говоря, в этой книге под названием *JavaScript* подразумевается язык программирования, определяемый третьей и пятой версиями стандарта ECMA-262. Этот стандарт дает определение языка программирования ECMAScript, который более известен в различных веб-браузерах по его реализациям JavaScript и JScript.

Кроме того, стандарты ECMA-262v3 и ECMA-262v5 дают определение диалектов регулярных выражений и замещающего текста, используемых в языке JavaScript. В данной книге эти диалекты обозначаются названием «JavaScript».

XRegExp

XRegExp – это открытая библиотека JavaScript, разработанная Стивеном Левитаном (Steven Levithan). Получить ее можно по адресу *http://xregexp.com*. Библиотека XRegExp расширяет синтаксис регулярных выражений диалекта JavaScript. Она также предоставляет функции, замещающие функции сопоставления регулярных выражений на языке JavaScript, скрывая некоторые несовместимости браузеров, и несколько новых высокоуровневых функций, упрощающих обход всех найденных совпадений.

В большинстве рецептов в этой главе мы не будем отделять решения для JavaScript и XRegExp. Вы можете использовать стандартные решения для JavaScript с регулярными выражениями, созданными с применением XRegExp. В ситуациях, когда применение библиотеки XRegExp обеспечивает более удачное решение, мы будем показывать код не только для стандартного JavaScript, но и для JavaScript с библиотекой XRegExp.

PHP

В языке PHP имеется три набора функций, предназначенных для работы с регулярными выражениями. Мы настоятельно рекомендуем использовать семейство функций `preg`. В этой книге рассматриваются только функции семейства `preg`, которые были встроены в язык PHP начиная с версии 4.2.0. В этой книге охватываются версии PHP 4 и 5. Функции `preg` в языке PHP являются обертками вокруг функций библиотеки PCRE. Диалект регулярных выражений PCRE в этой книге обозначается «PCRE». Поскольку в библиотеке PCRE отсутствуют функции, реализующие поиск с заменой, разработчиками

языка PHP был выработан собственный синтаксис определения замещающего текста для функции `preg_replace`. Данный диалект замещающего текста в этой книге обозначается как «PHP».

Семейство функций `mb_ereg` является составной частью семейства «многобайтовых» функций языка PHP, предназначенных для работы с текстом на языках, для которых традиционно используется многобайтовое представление символов, таких как японский и китайский. В версии PHP 5 функции семейства `mb_ereg` используют библиотеку поддержки регулярных выражений Onigurama, которая изначально предназначалась для языка программирования Ruby. Диалект регулярных выражений Onigurama в этой книге обозначается как «Ruby 1.9». Использовать функции семейства `mb_ereg` рекомендуется только в случае необходимости работы с многобайтовыми представлениями символов при условии, что вы уже знакомы с функциями `mb_` языка PHP.

Семейство функций `ereg` является старейшим семейством функций поддержки регулярных выражений в языке PHP и официально не рекомендуется к использованию начиная с версии PHP 5.3.0. Эти функции не зависят от внешних библиотек и реализуют диалект POSIX ERE. Данный диалект поддерживает лишь ограниченный набор возможностей и в этой книге не рассматривается. Диалект POSIX ERE – это ограниченное подмножество диалектов Ruby 1.9 и PCRE. Регулярные выражения, применяемые в вызовах функций `ereg`, можно использовать в функциях `mb_ereg` и `preg`. При этом в случае функций `preg` придется добавить разделители в стиле языка Perl (рецепт 3.1).

Perl

То, что язык Perl имеет встроенную поддержку регулярных выражений, является главной причиной их популярности в наши дни. Диалекты регулярных выражений и замещающего текста, используемые операторами `m//` и `s///` языка Perl, в этой книге обозначаются названием «Perl». В этой книге рассматриваются версии Perl 5.6, 5.8, 5.10 и 5.14.

Python

Язык программирования Python обеспечивает поддержку регулярных выражений посредством модуля `re`. Диалекты регулярных выражений и замещающего текста, используемые этим модулем, в данной книге обозначаются как «Python». В этой книге рассматриваются версии Python от 2.4 до 3.2.

Ruby

Поддержка регулярных выражений в языке Ruby реализована как часть самого языка. В этой книге рассматриваются версии Ruby 1.8 и 1.9. В этих двух версиях Ruby по умолчанию используются разные реализации механизма регулярных выражений. В версии Ruby 1.9 по умолчанию используется механизм Onigurama, обладающий более широкими возможностями, чем классический механизм в версии

Ruby 1.8. Более подробно об этом рассказывается в разделе «Диалекты регулярных выражений, рассматриваемые в этой книге» (глава 1).

В этой главе мы не будем углубляться в различия между версиями Ruby 1.8 и 1.9. Здесь демонстрируются лишь самые простые регулярные выражения, в которых не используются новые особенности версии Ruby 1.9. Поскольку поддержка регулярных выражений включена непосредственно в язык Ruby, программный код на этом языке, использующий регулярные выражения, остается неизменным независимо от того, скомпилирован Ruby с поддержкой классического механизма регулярных выражений или механизма Onigurama. В случае необходимости версия Ruby 1.8 может быть скомпилирована с поддержкой механизма Onigurama.

Другие языки программирования

Языки программирования, перечисленные ниже, не рассматриваются в этой книге, но они используют тот или иной диалект регулярных выражений из описываемых здесь. Если вы используете один из этих языков, можете просто пропустить эту главу, при этом другие главы по-прежнему будут для вас полезны.

ActionScript

Язык программирования ActionScript – это реализация стандарта ECMA-262, выполненная компанией Adobe. Начиная с версии 3.0 язык ActionScript полностью поддерживает регулярные выражения стандарта ECMA-262v3. Данный диалект регулярных выражений в этой книге обозначается как «JavaScript». Кроме того, язык ActionScript очень близок к языку JavaScript. Примеры из этой книги, которые приводятся для языка JavaScript, легко могут быть адаптированы под язык ActionScript.

C

В программах на языке C можно использовать широкий круг библиотек поддержки регулярных выражений. Из диалектов, рассматриваемых в книге, наилучшим, пожалуй, выбором будет библиотека PCRE, распространяемая с открытыми исходными текстами. Загрузить исходные тексты библиотеки на языке C можно на сайте *http://www.pcre.org*. Программный код библиотеки написан так, что позволяет компилировать его с помощью самых разных компиляторов на самых разных платформах.

C++

В программах на языке C++ можно использовать широкий круг библиотек поддержки регулярных выражений. Из диалектов, рассматриваемых в книге, наилучшим, пожалуй, выбором будет библиотека PCRE, распространяемая с открытыми исходными текстами. В программах можно использовать C API библиотеки напрямую или прибегнуть к помощи классов-оберток на языке C++, включенных

в состав PCRE и доступных для загрузки (подробности смотрите на сайте *http://www.pcre.org*).

В Windows можно импортировать COM-объект RegExp из VBScript 5.5, как описывается далее в подразделе с описанием Visual Basic 6. Это может пригодиться для обеспечения непротиворечивости диалектов регулярных выражений между серверными компонентами на C++ и клиентскими сценариями на JavaScript.

Стандарт C++ TR1 определяет заголовочный файл <regex>, содержащий определения функций, таких как `regex_search()`, `regex_match()` и `regex_replace()`, которые можно использовать для организации поиска в строках, проверки строк и поиска с заменой в строках с применением регулярных выражений. Поддержка регулярных выражений, определяемая стандартом C++ TR1, основана на библиотеке Boost.Regex. Если ваш компилятор C++ не поддерживает стандарт TR1, вы можете использовать саму библиотеку Boost.Regex. Полный комплект документации к этой библиотеке можно найти по адресу *http://www.boost.org/libs/regex/*.

Delphi

Версия Delphi XE стала первой, получившей встроенную поддержку регулярных выражений. В последующей версии Delphi XE2 возможности регулярных выражений никак не изменились. Модуль `RegularExpressionsAPI`, реализующий эту поддержку, является тонкой оберткой вокруг библиотеки PCRE. Скорее всего, вам не придется использовать этот модуль непосредственно.

Модуль `RegularExpressionsCore` реализует класс `TPerlRegEx`. Он предоставляет полный комплект методов для поиска, замены и разбиения строк с использованием регулярных выражений. Для представления строк он использует тип `UTF8String`, так как библиотека PCRE основана на применении кодировки UTF-8. Класс `TPerlRegEx` удобно использовать в ситуациях, когда требуется полный контроль над преобразованием строк в/из кодировки UTF-8 или когда исходные данные уже представлены текстом в кодировке UTF-8. Этот модуль можно также применять для переноса программного кода, написанного для старых версий Delphi и использующего класс `TPerlRegEx`, написанный Яном Гойвертсом. Модуль `RegularExpressionsCore` основан на наработках Яна, переданных им компании Embarcadero.

В новом коде вы, скорее всего, будете использовать модуль `RegularExpressions`. Он реализует записи, такие как `TRegex` и `TMatch`, имеющие имена и методы, близко имитирующие классы регулярных выражений в .NET Framework. Так как это обычные записи, вам не придется явно создавать и уничтожать их экземпляры. Они предоставляют массу статических методов, позволяющих использовать регулярные выражения в однострочных инструкциях.

Если вам приходится работать с более старыми версиями Delphi, рекомендуем использовать компонент `TPerlRegEx`, написанный Яном

Гойвертсом. Загрузить его можно по адресу *http://www.regexp.info/delphi.html*. Это открытый компонент, распространяемый на условиях лицензии Mozilla Public License. Последняя версия `TPerlRegEx` полностью совместима с модулем `RegularExpressionsCore` в Delphi XE. Для новых программ, которые создаются в версиях Delphi 2010 или ниже, мы настоятельно рекомендуем использовать `TPerlRegEx`. Если позднее вам придется переносить свой код на Delphi XE, вам останется всего лишь заменить `PerlRegEx` на `RegularExpressionsCore` в предложениях `uses`. При компиляции в версиях Delphi 2009 и Delphi 2010 модуль `PerlRegEx` использует тип `UTF8String` и обеспечивает полную поддержку Юникода. При компиляции в версии Delphi 2007 или ниже для представления строк используется тип `AnsiString` и Юникод не поддерживается.

Еще одной популярной оберткой вокруг библиотеки PCRE для Delphi является класс `TJclRegEx`, входящий в состав библиотеки JCL (*http://www.delphi-jedi.org*). Эта библиотека также распространяется с открытыми исходными текстами на условиях лицензии Mozilla Public License.

Delphi Prism

В Delphi Prism можно использовать поддержку регулярных выражений, предоставляемую платформой .NET Framework. Для этого достаточно добавить модуль `System.Text.RegularExpressions` в предложение `uses` в любом модуле Delphi Prism, где предполагается применять регулярные выражения.

После этого можно будет использовать те же приемы, которые демонстрируются в этой главе во фрагментах программного кода на языках C# и VB.NET.

Groovy

В программах на языке Groovy можно использовать регулярные выражения, поддержка которых обеспечивается пакетом `java.util.regex`, как в языке Java. Все решения, представленные в этой главе для языка Java, также должны работать и в языке Groovy. Собственный синтаксис регулярных выражений в языке Groovy просто обеспечивает сокращенную форму записи. Литерал регулярного выражения, части которого отделяются друг от друга символами слэша, является экземпляром класса `java.lang.String`, а оператор `=~` создает экземпляр класса `java.util.regex.Matcher`. Вы свободно можете смешивать синтаксис языка Groovy со стандартным синтаксисом языка Java – классы и объекты при этом остаются теми же самыми.

PowerShell

Язык PowerShell – это язык сценариев, разработанный компанией Microsoft и опирающийся на платформу .NET Framework. Операторы `-match` и `-replace`, встроенные в PowerShell, используют диалект регулярных выражений и замещающего текста .NET, который описывается в этой книге.

R

Проект R поддерживает регулярные выражения с помощью функций `grep`, `sub` и `regexpr` в пакете `base`. Все эти функции принимают необязательный параметр с именем `perl`, который получает значение `FALSE`, если он опущен. Если передать в нем значение `TRUE`, будет использоваться диалект PCRE, описываемый в этой книге. Регулярные выражения, демонстрируемые для версии PCRE 7, будут работать в версии R 2.5.0 и выше. В более ранних версиях R следует использовать регулярные выражения, помеченные в этой книге как «PCRE 4 и выше». Диалекты «базовый» и «расширенный», поддерживаемые языком R, являются устаревшими диалектами регулярных выражений с ограниченными возможностями и не рассматриваются в этой книге.

REALbasic

Язык REALbasic имеет встроенный класс `RegEx`. Реализация этого класса опирается на версию библиотеки PCRE с поддержкой кодировки UTF-8. Это означает, что имеется возможность задействовать поддержку Юникода в библиотеке PCRE, но при этом придется использовать класс `TextConverter` языка REALbasic для преобразования текста в кодировку UTF-8, прежде чем передавать его классу `RegEx`.

Все регулярные выражения, демонстрируемые в этой книге для версии PCRE 7, будут работать и в программах на языке REALbasic. Единственная особенность REALbasic состоит в том, что параметры «нечувствительность к регистру символов» (`Regex.Options.CaseSensitive`) и «символам ^ и $ соответствуют границы строк» (`Regex.Options.TreatTargetAsOneLine`) включены по умолчанию. Если появится необходимость использовать регулярные выражения из этой книги, для которых явно не говорится, что они используют эти режимы сопоставления, их следует явно отключить в программе на языке REALbasic.

Scala

Язык Scala обладает встроенной поддержкой регулярных выражений посредством модуля `scala.util.matching`. Эта поддержка построена на основе механизма регулярных выражений пакета `java.util.regex` языка Java. Диалекты регулярных выражений и замещающего текста, используемые в языках Java и Scala, обозначаются в этой книге как «Java».

Visual Basic 6

Версия Visual Basic 6 – это последняя версия Visual Basic, которая не требует наличия платформы .NET Framework. Это также означает, что Visual Basic 6 не использует превосходную поддержку регулярных выражений, реализованную в платформе .NET Framework. Примеры программного кода на языке VB.NET, которые приводятся в этой книге, вообще не будут работать в VB 6.

Visual Basic 6 обеспечивает простоту использования функциональных возможностей, предоставляемых библиотеками ActiveX и COM. Одной из таких библиотек является библиотека Microsoft VBScript, которая начиная с версии 5.5 обладает неплохой поддержкой регулярных выражений. Библиотека реализует тот же диалект регулярных выражений, что используется в языке JavaScript, описываемый стандартом ECMA-262v3. Данная библиотека является частью Internet Explorer 5.5 и выше. Она доступна на любом компьютере, работающем под управлением операционной системы Windows XP или Vista, а также в предыдущих версиях Windows при условии, что там установлена версия Internet Explorer 5.5 или выше. В эту категорию входят практически все персональные компьютеры, работающие под управлением Windows и используемые для подключения к Интернету.

Чтобы использовать эту библиотеку в приложении на языке Visual Basic, нужно выбрать пункт меню Project|References в среде разработки VB. Прокрутить список и отыскать в нем пункт «Microsoft VBScript Regular Expressions 5.5», расположенный сразу же под пунктом «Microsoft VBScript Regular Expressions 1.0». Убедитесь, что выбрали версию 5.5, а не 1.0. Версия 1.0 предоставляется только для сохранения обратной совместимости, а ее возможности менее чем удовлетворительны.

После добавления ссылки можно будет увидеть, какие классы предоставляются библиотекой и какие члены входят в состав этих классов. Для этого нужно выбрать пункт меню View|Object Browser. В окне Object Browser и в раскрывающемся списке в левом верхнем углу нужно выбрать библиотеку «VBScript_RegExp_55».

3.1. Литералы регулярных выражений в исходных текстах

Задача

Имеется регулярное выражение ‹[$"'\n\d/\\]›, являющееся решением некоторой задачи. Это регулярное выражение представляет собой символьный класс, которому соответствуют знак доллара, кавычка, апостроф, символ перевода строки, любая цифра в диапазоне от 0 до 9, символ слэша или символ обратного слэша. Необходимо вставить это регулярное выражение в исходный текст программы в виде строковой константы или оператора регулярного выражения.

Решение

C#

В виде обычной строки:

```
"[$\"'\n\\d/\\\\]"
```

В виде строкового литерала:

```
@"[$""'\n\d/\\]"
```

VB.NET

```
"[$""'\n\d/\\]"
```

Java

```
"[$\"'\n\\d/\\\\]"
```

JavaScript

```
/[$"'\n\d\/\\]/
```

XRegExp

```
"[$\"'\n\\d/\\\\]"
```

PHP

```
'%[$"\'\n\d/\\\\]%'
```

Perl

Оператор сопоставления с шаблоном:

```
/[\$"'\n\d\/\\]/
m![\$"'\n\d/\\]!
```

Оператор подстановки:

```
s![\$"'\n\d/\\]!!
```

Python

Строка в тройных кавычках:

```
r"""[$"'\n\d/\\]"""
```

Обычная строка:

```
"[$\"'\n\\d/\\\\]"
```

Ruby

Литерал регулярного выражения, разделенный символами слэша:

```
/[$"'\n\d\/\\]/
```

Литерал регулярного выражения, разделенный знаками пунктуации, выбираемыми пользователем:

```
%r![$"'\n\d/\\]!
```

Обсуждение

Когда в этой книге демонстрируется регулярное выражение само по себе (в противоположность регулярному выражению, входящему в состав программного кода), оно всегда приводится в чистом виде. Данный рецепт является единственным исключением. При использовании программы тестирования регулярных выражений, такой как RegexBuddy или RegexPal, регулярное выражение можно было бы ввести в таком виде. Если некоторое приложение принимает регулярные выражения, вводимые пользователем, пользователь должен был бы вводить его в таком виде.

Но если нужно поместить регулярное выражение в исходный текст программы, придется выполнить дополнительные действия. Небрежность, проявленная при копировании регулярного выражения из программы тестирования регулярных выражений в исходный текст вашей программы или обратно, часто может заставить поломать голову над тем, почему оно работает в программе тестирования, но не работает в исходном тексте программы. Или почему программа тестирования сообщает об ошибке в регулярном выражении, которое было скопировано из чьего-то программного кода. Все языки программирования, рассматриваемые в этой книге, требуют, чтобы литерал регулярного выражения был разграничен определенным способом. В одних языках регулярные выражения должны быть оформлены как строки, а в других – как специальные константы регулярных выражений. Если регулярное выражение содержит символы, которые являются разделителями или имеют специальное значение в конкретном языке программирования, их необходимо экранировать.

Наиболее часто в качестве экранирующего символа используется символ обратного слэша. Именно по этой причине большинство решений этой задачи включают гораздо большее число обратных слэшей по сравнению с оригинальными регулярными выражениями.

C#

В языке C# литерал регулярного выражения может передаваться конструктору `Regex()` и различным функциям-членам класса `Regex`. Параметр, в котором передается регулярное выражение, всегда объявляется как строка.

Язык C# поддерживает два типа строковых литералов. Наиболее часто используются строки в кавычках, хорошо известные по таким языкам программирования, как C++ и Java. Внутри таких строк кавычки и обратные слэши должны экранироваться символом обратного слэша. В строках также допускается указывать экранированные последовательности, обозначающие неотображаемые символы, такие как ‹\n›. При использовании свойства `RegexOptions.IgnorePatternWhitespace` (рецепт 3.4) для включения режима свободного форматирования, описываемого в ре-

цепте 2.18, наблюдаются различия между "\n" и "\\n". "\n" – это строка с символом перевода строки, который игнорируется как пробельный символ. "\\n" – это строка, включающая метасимвол <\n> регулярного выражения, совпадающий с символом перевода строки.

Строковые литералы начинаются со знака @ и кавычки и заканчиваются кавычкой. Чтобы включить кавычку в строковый литерал, ее необходимо продублировать. Символы обратного слэша не требуется экранировать, в результате регулярное выражение получается гораздо более удобочитаемым. @"\n" – это всегда метасимвол <\n> регулярного выражения, который совпадает с символом перевода строки, даже в режиме свободного форматирования. Строковые литералы не поддерживают <\n> на уровне строки, зато они могут располагаться на нескольких строках. Это делает строковые литералы идеальными для представления регулярных выражений в режиме свободного форматирования.

Выбор очевиден: чтобы поместить регулярное выражение в исходный текст программы на языке C#, предпочтительнее использовать строковые литералы.

VB.NET

В языке VB.NET литерал регулярного выражения может передаваться конструктору `Regex()` и различным функциям-членам класса `Regex`. Параметр, в котором передается регулярное выражение, всегда объявляется как строка.

В Visual Basic используются строки в кавычках. Кавычки внутри строк должны дублироваться. Экранировать какие-либо другие символы в них не требуется.

Java

В языке Java литерал регулярного выражения может передаваться фабричному методу класса `Pattern.compile()` и различным функциям класса `String`. Параметр, в котором передается регулярное выражение, всегда объявляется как строка.

В Java используются строки в кавычках. Кавычки и символы обратного слэша внутри строк должны экранироваться символом обратного слэша. В строках также допускается указывать экранированные последовательности, обозначающие неотображаемые символы, такие как <\n>, и кодовые пункты Юникода, такие как <\uFFFF>.

При использовании свойства `Pattern.COMMENTS` (рецепт 3.4) для включения режима свободного форматирования, описываемого в рецепте 2.18, наблюдаются различия между "\n" и "\\n". "\n" – это строка с символом перевода строки, который игнорируется как пробельный символ. "\\n" – это строка, включающая метасимвол <\n> регулярного выражения, совпадающий с символом перевода строки.

JavaScript

В языке JavaScript регулярные выражения лучше всего создавать с использованием специального синтаксиса объявления литералов регулярных выражений. Для этого нужно просто поместить регулярное выражение между двумя символами слэша. Если символы слэша присутствуют в самом регулярном выражении, их следует экранировать символом обратного слэша.

Несмотря на то что имеется возможность создавать объект `RegExp` из строки, нет смысла использовать строковую форму записи для представления регулярных выражений в программном коде. В этом случае потребовалось бы экранировать кавычки и символы обратного слэша, что обычно приводит к появлению частокола из обратных слэшей.

XRegExp

При использовании библиотеки XRegExp для расширения синтаксиса регулярных выражений JavaScript все объекты `XRegExp` создаются из строк, в которых необходимо экранировать кавычки, апострофы и обратные слэши.

PHP

Синтаксис литералов регулярных выражений, используемый функциями `preg` в языке PHP, представляет собой довольно хитроумное изобретение. В отличие от JavaScript и Perl, в языке отсутствует специализированный тип регулярного выражения. Регулярные выражения всегда должны быть представлены в виде строк. То же самое относится к функциям `ereg` и `mb_ereg`. Но в своем стремлении к подражанию языку Perl разработчики функций-оберток для библиотеки PCRE в языке PHP добавили дополнительное требование.

Внутри строки регулярное выражение должно записываться как литерал регулярного выражения в языке Perl. Это означает, что там, где в языке Perl регулярное выражение записывается как `/regex/`, при передаче его функциям `preg` в языке PHP оно должно записываться как `'/regex/'`. Как и в языке Perl, в качестве разделителей допускается использовать любую пару знаков пунктуации. Если символ-разделитель присутствует внутри регулярного выражения, его необходимо экранировать символом обратного слэша. Чтобы избежать этого, следует выбирать символ-разделитель, отсутствующий внутри регулярного выражения. В этом рецепте использовался знак процента, потому что символ слэша присутствует в регулярном выражении, а знак процента – нет. Если символ слэша отсутствует в регулярном выражении, можно использовать его, так как это наиболее часто используемый разделитель в языке Perl и обязательный разделитель в языках JavaScript и Ruby.

Язык PHP поддерживает строки в кавычках и в апострофах. В обоих случаях необходимо экранировать кавычки (или апострофы), а также

обратные слэши, присутствующие в регулярном выражении, символом обратного слэша. При использовании строк в кавычках дополнительно требуется экранировать знак доллара. Для записи регулярных выражений предпочтительнее использовать строки в апострофах, если не требуется интерпретировать переменные внутри регулярных выражений.

Perl

В языке Perl литералы регулярных выражений используются в операторе сопоставления с шаблоном и в операторе подстановки. Оператор сопоставления с шаблоном содержит два символа слэша и регулярное выражение, заключенное между ними. Символы слэша внутри регулярного выражения должны экранироваться символами обратного слэша. Какие-либо другие символы экранировать не требуется, за исключением, возможно, символов `$` и `@`, по причинам, описываемым в конце этого раздела.

Существует альтернативная форма записи оператора сопоставления с шаблоном, когда регулярное выражение помещается между парой любых других знаков пунктуации; в этом случае оператор должен начинаться с символа `m`. При использовании в качестве разделителей любых типов открывающих и закрывающих знаков пунктуации (круглые, квадратные или фигурные скобки) они должны соответствовать друг другу, например `m{`*regex*`}`. Если используется другой знак пунктуации, следует просто использовать его дважды. В решении к этому рецепту используется восклицательный знак. Это позволило сэкономить на экранировании символов слэша в регулярном выражении. Если в качестве разделителей используются знаки пунктуации, имеющие открывающую и закрывающую формы, то экранировать необходимо только закрывающий знак пунктуации, если он используется в регулярном выражении как литерал.

Оператор подстановки похож на оператор сопоставления с шаблоном. Только вместо символа `m` он начинается с символа `s` и принимает замещающий текст. При использовании квадратных скобок или подобных им знаков пунктуации необходимо использовать две пары: `s[`*regex*`] [`*replace*`]`. При смешивании разных разделителей также необходимо использовать две пары: `s[`*regex*`]/`*replace*`/`. Во всех остальных случаях знак пунктуации используется трижды: `s/`*regex*`/`*replace*`/`.

Операторы сопоставления с шаблоном и подстановки в языке Perl интерпретируются как строки в кавычках. Если регулярное выражение записано как `‹m/I•am•$name/›` и переменная `$name` хранит строку `"Jan"`, в результате будет получено регулярное выражение `‹I•am•Jan›`. Кроме того, в языке Perl `$"` – это переменная, поэтому потребовалось экранировать знак доллара в символьном классе регулярного выражения для этого рецепта.

Никогда не следует экранировать знак доллара, который используется как якорный метасимвол (рецепт 2.5). Экранированный знак доллара

всегда интерпретируется как литерал. Язык Perl в состоянии отличать знаки доллара, используемые как якорные метасимволы, от знаков доллара, используемых для интерпретации переменных, имея в виду тот факт, что якорный метасимвол может использоваться только в конце группы, или в конце всего регулярного выражения, или перед символом перевода строки. Не требуется экранировать знак доллара в выражении `‹m/^regex$/›`, если требуется проверить, соответствует ли регулярному выражению «regex» строка целиком.

Символ @ не имеет специального назначения в регулярных выражениях, но в языке Perl он используется для интерпретации переменных. Его необходимо экранировать в литералах регулярных выражений, включенных в программный код на языке Perl, как если бы это были строки в кавычках.

Python

Функции модуля `re` в языке Python принимают литералы регулярных выражений в виде строк. Можно использовать любые способы определения строк, которые допускаются языком Python. В зависимости от того, какие символы встречаются в регулярном выражении, различные формы записи строк могут уменьшить необходимость экранирования с помощью символов обратного слэша.

Обычно наилучшим выбором являются «сырые» (raw) строки. Сырые строки в языке Python вообще не требуют экранирования каких-либо символов. В случае использования сырых строк отпадает необходимость дублировать символы обратного слэша в регулярных выражениях. Строка `r"\d+"` читается проще, чем `"\\d+"`, особенно если регулярное выражение длинное.

Единственный случай, когда сырые строки перестают быть лучшим выбором, – если регулярное выражение включает в себя кавычки и апострофы. Невозможно использовать сырую строку, заключенную в пару апострофов или кавычек, так как нет никакой возможности экранировать кавычки внутри регулярного выражения. В такой ситуации могут помочь строки в тройных кавычках, как это сделано в решении для языка Python в данном рецепте. Обычная строка показана в решении для сравнения.

Если в регулярном выражении необходимо использовать возможности поддержки Юникода, описанные в рецепте 2.7, то в Python 2.x надлежит использовать строки Юникода. Можно превратить обычную строку в строку Юникода, добавив перед открывающей кавычкой символ `u`. В Python 3.0 и выше все строки автоматически интерпретируются как строки Юникода.

Сырые строки не поддерживают возможность определения неотображаемых символов, таких как `\n`. Они интерпретируют экранированные последовательности как последовательность литералов. Это не является проблемой для модуля `re`. Он поддерживает эти экранированные

последовательности как часть синтаксиса регулярных выражений, а также как часть синтаксиса замещающего текста. Литерал \n в сырой строке по-прежнему будет интерпретироваться как символ перевода строки в регулярных выражениях и в замещающем тексте.

При использовании атрибута re.VERBOSE (рецепт 3.4) для включения режима свободного форматирования, описываемого в рецепте 2.18, наблюдаются различия между строкой "\n" с одной стороны и строкой "\\n" и сырой строкой r"\n" – с другой. "\n" – это строка с символом перевода строки, который игнорируется как пробельный символ. "\\n" и r"\n" – это строки, включающие метасимвол ‹\n› регулярного выражения, совпадающий с символом перевода строки.

При использовании режима свободного форматирования лучшим выбором являются сырые строки в тройных кавычках, такие как r"""\n""", потому что они могут располагаться на нескольких строках. Кроме того, ‹\n› не интерпретируется на уровне строки, поэтому он может интерпретироваться на уровне регулярного выражения как метасимвол, совпадающий с границей строки.

Ruby

В языке Ruby регулярные выражения лучше всего создавать с использованием специального синтаксиса объявления регулярных выражений. Для этого нужно просто поместить регулярное выражение между двумя символами слэша. Если символ слэша присутствует внутри самого регулярного выражения, его следует экранировать символом обратного слэша.

Если нежелательно экранировать символы слэша в регулярном выражении, можно поставить перед началом регулярного выражения префикс %r и затем использовать в качестве разделителя любой другой символ пунктуации.

Несмотря на то что существует возможность создавать объекты Regexp из строки, тем не менее нет смысла использовать строковую форму записи для представления регулярных выражений в программном коде. В этом случае потребовалось бы экранировать кавычки и символы обратного слэша, что обычно приводит к появлению частокола из обратных слэшей.

В этом отношении язык Ruby очень напоминает язык JavaScript, за исключением имени класса, которое в языке Ruby записывается одним словом Regexp, а в языке JavaScript – как RegExp (в стиле «camel caps»).

См. также

Рецепт 2.3, где объясняется, как работают символьные классы и почему в регулярных выражениях необходимо записывать два символа об-

ратного слэша, чтобы включить только один символ в символьный класс.

Рецепт 3.4, где описывается, как установить параметры регулярного выражения, которые в отдельных языках программирования реализованы как часть литералов регулярных выражений.

3.2. Импортирование библиотеки регулярных выражений

Задача

Прежде чем появится возможность использовать регулярные выражения в приложении, необходимо импортировать в программный код библиотеку регулярных выражений или пространство имен.

В остальных фрагментах исходных текстов в этой книге предполагается, что импортирование уже было выполнено, если оно было необходимо.

Решение

C#

```
using System.Text.RegularExpressions;
```

VB.NET

```
Imports System.Text.RegularExpressions
```

XRegExp

В коде JavaScript, выполняющемся в браузере:

```
<script src="xregexp-all-min.js"></script>
```

В коде JavaScript, выполняющемся на сервере под управлением Node.js:

```
var XRegExp = require('xregexp').XRegExp;
```

Java

```
import java.util.regex.*;
```

Python

```
import re
```

Обсуждение

В некоторых языках программирования поддержка регулярных выражений встроена в сам язык. При использовании этих языков не требу-

ется импортировать что-либо, чтобы обеспечить поддержку регулярных выражений. В других языках поддержка регулярных выражений обеспечивается с помощью библиотек, которые необходимо импортировать в исходных текстах. Некоторые языки программирования вообще не имеют поддержки регулярных выражений. Для таких языков программирования поддержку регулярных выражений необходимо скомпилировать и присоединить самостоятельно.

C#

Если поместить в начало файла с исходным программным кодом на языке C# инструкцию `using`, появится возможность напрямую обращаться к классам, обеспечивающим функциональные возможности регулярных выражений без необходимости полностью квалифицировать их имена. Например, обращение `System.Text.RegularExpressions.Regex()` можно будет записать просто как `Regex()`.

VB.NET

Если поместить в начало файла с исходным программным кодом на языке VB.NET инструкцию `Imports`, появится возможность напрямую обращаться к классам, обеспечивающим функциональные возможности регулярных выражений без необходимости полностью квалифицировать их имена. Например, обращение `System.Text.RegularExpressions.Regex()` можно будет записать просто как `Regex()`.

Java

Чтобы в языке Java воспользоваться встроенной библиотекой регулярных выражений, необходимо импортировать в приложение пакет `java.util.regex`.

JavaScript

Поддержка регулярных выражений в JavaScript встроена в язык и доступна всегда.

XRegExp

Чтобы получить возможность использовать в веб-браузере расширенный синтаксис регулярных выражений, предоставляемый библиотекой XRegExp, необходимо загрузить эту библиотеку. Для этого достаточно подключить к веб-странице минимизированную версию библиотеки `xregexp-all-min.js`, обеспечивающую полную функциональность XRegExp. В этой книге в решениях на основе XRegExp предполагается, что подключена эта версия библиотеки.

Если для вас время загрузки страницы имеет большое значение и вы не используете категории, блоки и/или алфавиты Юникода, используйте базовую версию библиотеки `xregexp-min.js` и загружайте дополнитель-

ные модули по мере необходимости. Например, чтобы добавить синтаксис ‹\p{…}› поддержки свойств Юникода, достаточно загрузить unicode-base.js. Далее можно загрузить unicode-blocks.js, unicode-categories.js и/или unicode-scripts.js, чтобы обеспечить возможность сопоставления с блоками, категориями и/или алфавитами Юникода с применением синтаксиса ‹\p{…}›.

Если вы используете Node.js для выполнения сценариев JavaScript на сервере, установите библиотеку XRegExp как пакет npm. Сделать это можно, выполнив команду npm install xregexp в командной строке. После установки серверные сценарии получат возможность импортировать библиотеку XRegExp, как показано в разделе «Решение».

PHP

Функции preg встроены в сам язык и всегда доступны в версии PHP 4.2.0 и выше.

Perl

Встроенная поддержка регулярных выражений в языке Perl доступна всегда.

Python

Чтобы получить возможность использовать функции регулярных выражений в языке Python, необходимо в сценарии импортировать модуль re.

Ruby

Встроенная поддержка регулярных выражений в языке Ruby доступна всегда.

3.3. Создание объектов регулярных выражений

Задача

Необходимо создать объект регулярного выражения или как-то иначе скомпилировать регулярное выражение, чтобы его можно было эффективно использовать в приложении.

Решение

C#

Если известно, что регулярное выражение не содержит ошибок:

```
Regex regexObj = new Regex("regex pattern");
```

Если регулярное выражение предоставляется конечным пользователем (ввод пользователя сохранен в строковой переменной UserInput):

```
try {
    Regex regexObj = new Regex(UserInput);
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

Если известно, что регулярное выражение не содержит ошибок:

```
Dim RegexObj As New Regex("regex pattern")
```

Если регулярное выражение предоставляется конечным пользователем (ввод пользователя сохранен в строковой переменной `UserInput`**):**

```
Try
    Dim RegexObj As New Regex(UserInput)
Catch ex As ArgumentException
     'Синтаксическая ошибка в регулярном выражении
End Try
```

Java

Если известно, что регулярное выражение не содержит ошибок:

```
Pattern regex = Pattern.compile("regex pattern");
```

Если регулярное выражение предоставляется конечным пользователем (ввод пользователя сохранен в строковой переменной `userInput`**):**

```
try {
    Pattern regex = Pattern.compile(userInput);
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

Чтобы иметь возможность использовать регулярное выражение в виде строки, необходимо создать объект `Matcher`**:**

```
Matcher regexMatcher = regex.matcher(subjectString);
```

Чтобы использовать регулярное выражение в виде другой строки, необходимо создать новый объект `Matcher`**, как только что было продемонстрировано, или повторно использовать существующий объект:**

```
regexMatcher.reset(anotherSubjectString);
```

JavaScript

Литерал регулярного выражения в программном коде:

```
var myregexp = /regex pattern/;
```

Регулярное выражение, полученное от пользователя и хранящееся в переменной `userinput`**:**

```
var myregexp = new RegExp(userinput);
```

XRegExp

Если в JavaScript потребуется использовать расширенный синтаксис библиотеки XRegExp, необходимо создать объект XRegExp из строки:

```
var myregexp = XRegExp("regex pattern");
```

Perl

```
$myregex = qr/regex pattern/
```

Регулярное выражение, полученное от пользователя и хранящееся в переменной $userinput:

```
$myregex = qr/$userinput/
```

Python

```
reobj = re.compile("regex pattern")
```

Регулярное выражение, полученное от пользователя и хранящееся в переменной userinput:

```
reobj = re.compile(userinput)
```

Ruby

Литерал регулярного выражения в программном коде:

```
myregexp = /regex pattern/;
```

Регулярное выражение, полученное от пользователя и хранящееся в переменной userinput:

```
myregexp = Regexp.new(userinput);
```

Обсуждение

Прежде чем механизм регулярных выражений сможет выполнить сопоставление регулярного выражения со строкой, оно должно быть скомпилировано. Компиляция выражения производится во время работы приложения. Конструктор объекта регулярного выражения или функция, выполняющая компилирование, производит синтаксический анализ строки, в которой хранится регулярное выражение, и преобразует ее в древовидную структуру или в конечный автомат. Функция, выполняющая фактическое сопоставление с шаблоном, выполняет обход этого дерева или конечного автомата в процессе просмотра строки с испытуемым текстом. В языках программирования, поддерживающих литералы регулярных выражений, компиляция выражения производится, когда поток выполнения достигает оператора регулярного выражения.

.NET

В языках C# и VB.NET скомпилированное регулярное выражение хранится в экземпляре класса System.Text.RegularExpressions.Regex. Самый

простой конструктор принимает всего один параметр: строку с регулярным выражением.

Если в регулярном выражении обнаруживаются синтаксические ошибки, конструктор `Regex()` возбуждает исключение `ArgumentException`. Сообщение исключения точно укажет, какая ошибка произошла. Если регулярные выражения вводятся пользователем приложения, очень важно предусмотреть обработку этого исключения, в ходе которой можно отобразить текст исключения и предложить пользователю исправить ошибку. Когда регулярное выражение включено в текст программы в виде строкового литерала, можно опустить действия по обработке исключения, если с помощью инструментальных средств анализа программного кода было проверено, что при выполнении строка не возбуждает исключение. Совершенно невозможно изменить состояние или режим так, чтобы один и тот же литерал регулярного выражения компилировался в одном состоянии и вызывал ошибку компиляции в другом. Обратите внимание, что в том случае если регулярное выражение содержит синтаксическую ошибку, исключение будет возбуждено на этапе выполнения программы, а не на этапе ее компиляции.

Если регулярное выражение предполагается использовать в цикле или многократно задействовать его в приложении, следует использовать объект `Regex`. Конструирование объекта регулярного выражения не создает дополнительной нагрузки. Статические члены класса `Regex`, которые принимают регулярное выражение в виде строкового параметра, в любом случае создают объект `Regex`, поэтому лучше сделать это в своем программном коде явно и потом хранить ссылку на созданный объект.

Если планируется использовать регулярное выражение один-два раза, можно пользоваться статическими членами класса `Regex`, чтобы сэкономить несколько строк программного кода. Статические члены класса `Regex` не стирают объект регулярного выражения, созданный внутренней реализацией, – они сохраняют в кэше 15 последних использовавшихся регулярных выражений. Существует возможность изменить размер кэша, установив значение свойства `Regex.CacheSize`. Поиск в кэше организован как поиск строки регулярного выражения. Но не следует во всем полагаться на кэш. Если в программе часто требуется большое число регулярных выражений, предпочтительнее будет создать собственный кэш, в котором можно организовать более эффективный поиск, чем поиск строки.

Java

В языке Java класс `Pattern` хранит одно скомпилированное регулярное выражение. Экземпляры этого класса можно создавать с помощью фабричной функции класса `Pattern.compile()`, которая принимает единственный параметр: строку с регулярным выражением.

Если в регулярном выражении обнаружится синтаксическая ошибка, функция `Pattern.compile()` возбудит исключение `PatternSyntaxException`.

Сообщение исключения точно укажет, какая ошибка произошла. Если регулярные выражения вводятся пользователем приложения, очень важно предусмотреть обработку этого исключения, в ходе которой можно отобразить текст исключения и предложить пользователю исправить ошибку. Когда регулярное выражение включено в текст программы в виде строкового литерала, можно опустить действия по обработке исключения, если с помощью инструментальных средств анализа программного кода было проверено, что при выполнении строка не возбуждает исключение. Совершенно невозможно изменить состояние или режим так, чтобы один и тот же литерал регулярного выражения компилировался в одном состоянии и вызывал ошибку компиляции в другом. Обратите внимание, что в случае если регулярное выражение содержит синтаксическую ошибку, исключение будет возбуждено на этапе выполнения программы, а не на этапе ее компиляции.

Если регулярное выражение предполагается использовать многократно, следует создать объект `Pattern` вместо того, чтобы использовать статические члены класса `String`. Хотя для этого потребуется несколько дополнительных строк программного кода, этот код будет работать гораздо эффективнее. Статические члены выполняют рекомпиляцию каждого регулярного выражения при каждом обращении. В Java статические члены реализованы исключительно для решения простейших задач применения регулярных выражений.

Объект `Pattern` просто хранит скомпилированную версию регулярного выражения – он не выполняет никакой фактической работы. Собственно сопоставление регулярного выражения производится классом `Matcher`. Чтобы создать экземпляр класса `Matcher`, необходимо вызвать метод `matcher()` объекта скомпилированного регулярного выражения. Метод `matcher()` принимает единственный аргумент – строку испытуемого текста.

Метод `matcher()` можно вызывать неограниченное число раз, чтобы применить одно и то же регулярное выражение к множеству строк. Можно работать с несколькими объектами, выполняющими сопоставление и использующими одно и то же регулярное выражение, при условии, что все они будут работать в одном потоке выполнения. Классы `Pattern` и `Matcher` не предусматривают возможность выполнения в многопоточной среде. Если необходимо использовать одно и то же регулярное выражение в нескольких потоках выполнения, функция `Pattern.compile()` должна вызываться в каждом таком потоке.

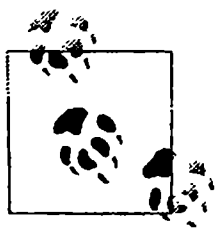
Если регулярное выражение, примененное к одной строке, нужно применить к другой строке, чтобы повторно использовать объект `Matcher`, необходимо вызвать его метод `reset()`, передав ему следующую строку с испытуемым текстом. Такой подход более эффективен, чем создание нового объекта `Matcher`. Метод `reset()` возвращает тот же объект `Matcher`, которому принадлежит метод, что позволяет производить повторную

инициализацию объекта и задействовать его в одной строке программного кода, например `regexMatcher.reset(nextString).find()`.

JavaScript

Форма записи регулярных выражений, как показано в рецепте 3.2, уже создает новый объект регулярного выражения. Чтобы повторно использовать тот же самый объект, нужно просто присвоить его переменной.

Если регулярное выражение хранится в строковой переменной (например, регулярное выражение, введенное пользователем), для его компиляции следует использовать конструктор `RegExp()`. Обратите внимание, что регулярное выражение внутри строки не окружается символами слэша. Эти слэши являются частью синтаксиса JavaScript записи объектов `RegExp` в виде литералов, а вовсе не частью самого регулярного выражения.



Так как запись литерала регулярного выражения в переменную является тривиальной операцией, в большинстве решений для JavaScript в этой главе эта строка кода опущена и напрямую используется литерал регулярного выражения. Если предполагается многократно применять одно и то же регулярное выражение, его необходимо присвоить переменной и использовать эту переменную, вместо того чтобы много раз вставлять один и тот же литерал регулярного выражения. Это повысит производительность и упростит сопровождение программного кода.

XRegExp

Если в сценарии на JavaScript потребуется использовать расширенный синтаксис регулярных выражений, предоставляемый библиотекой XRegExp, регулярные выражения следует компилировать вызовом конструктора `XRegExp()`. Для большей эффективности – когда одно и то же регулярное выражение предполагается использовать многократно – результат компиляции следует присвоить переменной и передавать эту переменную методам класса `XRegExp`, когда потребуется задействовать это регулярное выражение.

В ситуациях, когда невозможно организовать хранение переменной с объектом `XRegExp`, для компиляции регулярных выражений можно использовать метод `XRegExp.cache()`. Этот метод компилирует каждое регулярное выражение только один раз. При последующих вызовах метода с теми же параметрами он возвращает один и тот же ранее скомпилированный экземпляр `XRegExp`.

PHP

Язык PHP не предоставляет возможность сохранять скомпилированное регулярное выражение в переменной. Поэтому всякий раз когда возникает потребность выполнить какую-либо операцию с применением

регулярного выражения, функциям preg придется передавать регулярное выражение в виде строки.

Функции семейства preg хранят до 4096 скомпилированных регулярных выражений в своем кэше. Несмотря на то что поиск в кэше выполняется на основе хешей, скорость поиска намного ниже, чем скорость обращения к переменной, однако производительность от этого падает не так сильно, как она могла бы упасть, если бы одно и то же регулярное выражение пришлось компилировать снова и снова. Когда кэш заполняется, из него удаляется самое старое скомпилированное регулярное выражение.

Perl

В языке Perl имеется возможность с помощью оператора qr// скомпилировать регулярное выражение и присвоить его переменной. Этот оператор имеет тот же синтаксис, что и оператор сопоставления, описанный в рецепте 3.1, за исключением того, что он начинается не с символа m, а с символов qr.

Вообще язык Perl весьма эффективен в части многократного использования ранее скомпилированных регулярных выражений. Поэтому мы не будем использовать оператор qr// во фрагментах программного кода в этой главе. Его применение будет продемонстрировано только в рецепте 3.5.

Оператор qr// удобно использовать, когда необходимо интерпретировать переменные внутри регулярного выражения или когда само регулярное выражение является строкой (например, когда оно получено в результате ввода пользователем). При использовании qr/$regexstring/ имеется возможность управлять процессом повторной компиляции, чтобы отразить новое содержимое переменной $regexstring. Оператор m/$regexstring/ выполняет повторную компиляцию регулярного выражения всякий раз, тогда как m/$regexstring/o никогда не будет компилировать его повторно. Значение модификатора /o объясняется в рецепте 3.4.

Python

В языке Python в модуле re имеется функция compile(), которая принимает строку с регулярным выражением и возвращает объект со скомпилированным регулярным выражением.

Если то же самое регулярное выражение планируется использовать многократно, необходимо явно вызвать функцию compile(). Все функции, имеющиеся в модуле re, в самом начале вызывают функцию compile(), а затем вызывают требуемую функцию объекта скомпилированного регулярного выражения.

Функция compile() сохраняет ссылки на 100 последних скомпилированных регулярных выражений. Тем самым операция повторной компиляции любого из 100 последних использовавшихся регулярных вы-

ражений замещается операцией поиска по словарю. Когда кэш заполняется полностью, он очищается целиком.

Если проблема производительности не стоит слишком остро, быстродействия кэша будет вполне достаточно, чтобы напрямую использовать функции модуля `re`. Но когда производительность имеет большое значение, предпочтительнее применять функцию `compile()`.

Ruby

Форма записи регулярных выражений, продемонстрированная в рецепте 3.2, уже создает новый объект регулярного выражения. Чтобы повторно использовать тот же самый объект, нужно просто присвоить его переменной.

Если регулярное выражение хранится в строковой переменной (например, регулярное выражение, введенное пользователем), для его компиляции следует использовать фабричную функцию `Regexp.new()` или ее синоним `Regexp.compile()`. Обратите внимание, что регулярное выражение внутри строки не окружается символами слэша. Эти слэши являются частью синтаксиса Ruby записи объектов `Regexp` в виде литералов, а вовсе не частью самого регулярного выражения.

Так как запись литерала регулярного выражения в переменную является тривиальной операцией, в большинстве решений для Ruby в этой главе эта строка кода опущена и напрямую используется литерал регулярного выражения. Если предполагается многократно применять одно и то же регулярное выражение, его необходимо присвоить переменной и использовать эту переменную, вместо того чтобы много раз вставлять один и тот же литерал регулярного выражения. Это повысит производительность и упростит сопровождение программного кода.

Компилирование регулярных выражений до уровня языка CIL

C#

```
Regex regexObj = new Regex("regex pattern", RegexOptions.Compiled);
```

VB.NET

```
Dim RegexObj As New Regex("regex pattern", RegexOptions.Compiled)
```

Обсуждение

При создании объекта `Regex` на платформе .NET без передачи дополнительных параметров регулярное выражение компилируется именно так, как описывалось в разделе «Обсуждение» выше в этом же рецепте. Если вторым параметром конструктору `Regex()` передать значение

RegexOptions.Compiled, класс Regex выполнит нечто иное: он скомпилирует регулярное выражение до уровня языка CIL, известного также как MSIL. Название CIL происходит от «Common Intermediate Language» (общий промежуточный язык). Это низкоуровневый язык программирования, который гораздо ближе к языку ассемблера, чем к C# или Visual Basic. Все компиляторы .NET производят программный код на языке CIL. Когда приложение запускается первый раз, платформа .NET Framework компилирует программный код на языке CIL в машинные инструкции, подходящие для компьютера пользователя.

Преимущество компилирования регулярных выражений с параметром RegexOptions.Compiled заключается в том, что они могут выполняться до 10 раз быстрее, чем регулярные выражения, скомпилированные без этого параметра. Недостаток такого подхода состоит в том, что компиляция может выполняться на два порядка медленнее, чем простое преобразование строки с регулярным выражением в древовидную структуру. Кроме того, программный код на языке CIL становится постоянной частью приложения и сохраняется, пока приложение не завершит работу. Программный код CIL не убирается сборщиком мусора.

Параметр RegexOptions.Compiled следует использовать, только если регулярное выражение слишком сложное или необходимо обрабатывать большие объемы текста, когда задержки, возникающие во время выполнения операции с регулярным выражением, становятся слишком заметными для пользователя. Дополнительные затраты на компиляцию и сборку не имеют смысла, если регулярное выражение выполняет свою работу за доли секунды.

См. также

Рецепт 3.1, где разъясняется, как вставлять литералы регулярных выражений в исходные тексты программ.

Рецепт 3.2, где рассказывается, как импортировать библиотеки поддержки регулярных выражений в программах. В некоторых языках программирования необходимо выполнить этот шаг, чтобы получить возможность создавать объекты регулярных выражений.

Рецепт 3.4, где описывается, как установить параметры регулярного выражения, которые в отдельных языках программирования реализованы как часть литералов регулярных выражений.

3.4. Установка параметров регулярных выражений

Задача

Необходимо скомпилировать регулярное выражение со всеми доступными режимами сопоставления: «свободное форматирование», «нечув-

ствительность к регистру символов», «точке соответствуют границы строк» и «символам ^ и $ соответствуют границы строк».

Решение

C#

```
Regex regexObj = new Regex("regex pattern",
    RegexOptions.IgnorePatternWhitespace | RegexOptions.IgnoreCase |
    RegexOptions.Singleline | RegexOptions.Multiline);
```

VB.NET

```
Dim RegexObj As New Regex("regex pattern",
    RegexOptions.IgnorePatternWhitespace Or RegexOptions.IgnoreCase Or
    RegexOptions.Singleline Or RegexOptions.Multiline)
```

Java

```
Pattern regex = Pattern.compile("regex pattern",
    Pattern.COMMENTS | Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE |
    Pattern.DOTALL | Pattern.MULTILINE);
```

JavaScript

Литерал регулярного выражения в программном коде:

```
var myregexp = /regex pattern/im;
```

Регулярное выражение, полученное от пользователя в виде строки:

```
var myregexp = new RegExp(userinput, "im");
```

XRegExp

```
var myregexp = XRegExp("regex pattern", "xism");
```

PHP

```
regexstring = '/regex pattern/xism';
```

Perl

```
m/regex pattern/xism;
```

Python

```
reobj = re.compile("regex pattern",
    re.VERBOSE | re.IGNORECASE |
    re.DOTALL | re.MULTILINE)
```

Ruby

Литерал регулярного выражения в программном коде:

```
myregexp = /regex pattern/xim;
```

Регулярное выражение, полученное от пользователя в виде строки:

```
myregexp = Regexp.new(userinput,
    Regexp::EXTENDED or Regexp::IGNORECASE or
    Regexp::MULTILINE);
```

Обсуждение

Многие регулярные выражения из тех, что приводятся в книге, и из тех, что встретятся вам на практике, написаны с учетом использования в определенном режиме. Почти все современные диалекты регулярных выражений поддерживают четыре основных режима. К сожалению, в некоторых диалектах параметры, реализующие те или иные режимы, имеют противоречивые или запутывающие названия. Использование ошибочных режимов обычно отрицательно сказывается на работоспособности регулярного выражения.

Во всех решениях в этом рецепте для установки режимов используются флаги или параметры, предоставляемые языком программирования или классом регулярных выражений. Другой способ установки режимов заключается в использовании модификаторов режимов внутри регулярного выражения. Модификаторы режимов внутри регулярного выражения всегда имеют высший приоритет перед параметрами или флагами, которые устанавливаются за пределами регулярного выражения.

.NET

Конструктор `Regex()` принимает второй необязательный аргумент с параметрами регулярного выражения. Доступные параметры можно увидеть в перечислении `RegexOptions`.

Свободное форматирование: `RegexOptions.IgnorePatternWhitespace`

Нечувствительность к регистру символов: `RegexOptions.IgnoreCase`

Точке соответствуют границы строк: `RegexOptions.Singleline`

Символам ^ и $ соответствуют границы строк: `RegexOptions.Multiline`

Java

Фабричная функция `Pattern.compile()` принимает второй необязательный аргумент с параметрами регулярного выражения. Класс `Pattern` определяет несколько констант для установки различных режимов. Имеется возможность одновременно установить несколько параметров, объединив их оператором «поразрядное ИЛИ» `|`.

Свободное форматирование: `Pattern.COMMENTS`

Нечувствительность к регистру символов: `Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE`

Точке соответствуют границы строк: `Pattern.DOTALL`

Символам ^ и $ соответствуют границы строк: `Pattern.MULTILINE`

Имеется два параметра, определяющие режим нечувствительности к регистру символов, и для обеспечения полной нечувствительности к регистру символов следует установить оба. Если установить только `Pattern.CASE_INSENSITIVE`, сопоставление без учета регистра символов будет производиться только для латинских символов от A до Z. Если установить оба параметра, будут сопоставляться без учета регистра символов все символы из всех алфавитов. Единственная причина отказаться от использования параметра `Pattern.UNICODE_CASE` заключается в стремлении обеспечить высокую производительность, если заранее известно, что испытуемый текст содержит только символы ASCII. При использовании модификаторов режима внутри регулярных выражений модификатор `‹(?i)›` включает режим нечувствительности к регистру только для символов ASCII, а модификатор `‹(?iu)›` – режим полной нечувствительности к регистру символов.

JavaScript

Чтобы определить какие-либо параметры в языке JavaScript, необходимо добавить один или более односимвольных флагов в литерал объекта `RegExp` после символа слэша, завершающего регулярное выражение. В документации эти флаги обычно записываются как `/i` и `/m`, хотя флаг – это собственно буква. Перед флагами не должно добавляться никаких дополнительных символов слэша.

При использовании конструктора `RegExp()` для компиляции строки в объект регулярного выражения можно передать второй необязательный аргумент с флагами регулярного выражения. Второй аргумент должен быть строкой с символами параметров, которые требуется установить. В этой строке не должно присутствовать никаких символов слэша.

Свободное форматирование: не поддерживается в JavaScript

Нечувствительность к регистру символов: `/i`

Точке соответствуют границы строк: не поддерживается в JavaScript

Символам ^ и $ соответствуют границы строк: `/m`

XRegExp

Библиотека XRegExp расширяет синтаксис регулярных выражений в JavaScript, добавляя поддержку свободного форматирования и совпадения точки с границами строк, которым соответствуют флаги `"x"` и `"s"`, часто используемые в других диалектах регулярных выражений. Эти флаги можно передать конструктору `XRegExp()` в виде строки во втором параметре.

Свободное форматирование: `"x"`

Нечувствительность к регистру символов: `"i"`

Точке соответствуют границы строк: `"s"`

Символам ^ и $ соответствуют границы строк: `"m"`

PHP

В рецепте 3.1 говорилось, что функции семейства preg в языке PHP требуют, чтобы литерал регулярного выражения был заключен между двумя символами пунктуации, обычно символами слэша, и отформатирован как строковый литерал. Чтобы определить параметры регулярного выражения, нужно добавить в конец строки один или более односимвольных модификаторов. То есть символы модификаторов должны следовать за разделителем, закрывающим регулярное выражение, но оставаться внутри строки в апострофах или в кавычках. В документации эти флаги обычно записываются как /x, хотя любой флаг – это соответствующий символ, а разделитель между регулярным выражением и модификаторами необязательно должен быть символом слэша.

Свободное форматирование: /x

Нечувствительность к регистру символов: /i

Точке соответствуют границы строк: /s

Символам ^ и $ соответствуют границы строк: /m

Perl

Параметры регулярного выражения определяются добавлением одного или более односимвольных модификаторов в конец оператора сопоставления с шаблоном или подстановки. В документации эти флаги обычно записываются как /x, хотя любой флаг – это один символ, а разделитель между регулярным выражением и модификаторами необязательно должен быть символом слэша.

Свободное форматирование: /x

Нечувствительность к регистру символов: /i

Точке соответствуют границы строк: /s

Символам ^ и $ соответствуют границы строк: /m

Python

Функция compile() (описывается в предыдущем рецепте) принимает второй необязательный аргумент с параметрами регулярного выражения. Этот аргумент необходимо собирать с помощью оператора |, чтобы объединить значения констант, определяемых в модуле re. Многие другие функции в модуле re, которые принимают в качестве аргумента литерал регулярного выражения, также принимают параметры выражения в виде последнего необязательного аргумента.

Для каждого параметра регулярных выражений определена пара констант. То есть каждый параметр может быть представлен константой с полным именем параметра или константой с именем из одного символа. Значения констант внутри пары идентичны. Единственное отличие состоит в том, что использование полных имен делает программный код более удобочитаемым для других разработчиков, не знакомых с ал-

фавитом параметров регулярных выражений. Односимвольные имена констант параметров, перечисленные в этом разделе, соответствуют их аналогам в языке Perl.

Свободное форматирование: `re.VERBOSE` **или** `re.X`

Нечувствительность к регистру символов: `re.IGNORECASE` **или** `re.I`

Точке соответствуют границы строк: `re.DOTALL` **или** `re.S`

Символам ^ и $ соответствуют границы строк: `re.MULTILINE` **или** `re.M`

Ruby

Чтобы определить какие-либо параметры в языке Ruby, необходимо добавить один или более односимвольных флагов в литерал объекта `Regexp` после символа слэша, завершающего регулярное выражение. В документации эти флаги обычно записываются как `/i` и `/m`, хотя любой флаг – это один символ. Перед флагами не должно добавляться никаких символов слэша.

При использовании фабричной функции `Regexp.new()` для компиляции строки в объект регулярного выражения можно передать второй необязательный аргумент с флагами регулярного выражения. Второй аргумент должен быть либо значением `nil`, чтобы отключить все параметры, либо комбинацией констант класса `Regexp`, объединяемых оператором `or`.

Свободное форматирование: `/r` **или** `Regexp::EXTENDED`

Нечувствительность к регистру символов: `/i` **или** `Regexp::IGNORECASE`

Точке соответствуют границы строк: `/m` **или** `Regexp::MULTILINE`. В языке Ruby действительно используются модификатор «m» и название «multiline», тогда как в других диалектах для обозначения режима «точке соответствуют границы строк» используется модификатор «s» или название «singleline».

Символам ^ и $ соответствуют границы строк: символ крышки и знак доллара в языке Ruby всегда совпадают с границами строк. Этот режим невозможно отключить. Поэтому соответствие началу и концу строки с испытуемым текстом следует проверять с помощью метасимволов ‹\A› и ‹\Z›.

Дополнительные параметры, характерные для различных языков

.NET

Параметр `RegexOptions.ExplicitCapture` превращает все группы, за исключением именованных, в несохраняющие. С этим параметром конструкция ‹(…)› становится эквивалентной конструкции ‹(?:…)›. Если сохраняющие группы всегда оформляются как именованные, включение этого параметра обеспечит более высокую эффективность регулярного выражения без необходимости использовать синтаксическую конструк-

цию ‹(?:…)›. Вместо включения параметра RegexOptions.ExplicitCapture можно добавить в начало регулярного выражения модификатор режима ‹(?n)›. Подробнее о группировке рассказывается в рецепте 2.9, а об именованных группах рассказывается в рецепте 2.11.

Если регулярное выражение используется одновременно в .NET и в JavaScript и необходимо добиться одинакового его поведения, можно включить параметр RegexOptions.ECMAScript. Этот параметр особенно удобен при разработке клиентской части веб-приложения на JavaScript и серверной на ASP.NET. Наиболее важный эффект параметра состоит в том, что метасимволы ‹\w› и ‹\d› ограничиваются набором символов ASCII, как в языке JavaScript.

Java

В языке Java имеется уникальный параметр Pattern.CANON_EQ, который включает режим «канонической эквивалентности». Как уже говорилось в разделе «Графема Юникода» (рецепт 2.7), Юникод обеспечивает различные способы представления символов с диакритическими знаками. При включении этого параметра регулярное выражение будет совпадать с символами, даже если в испытуемом тексте они имеют различные представления. Например, регулярное выражение ‹\u00E0› будет совпадать и с последовательностью "\u00E0", и с последовательностью "\u0061\u0300", потому что они канонически эквивалентны. Оба представления выводятся на экран как символ «à» и неразличимы для конечного пользователя. При отключенном режиме канонической эквивалентности регулярное выражение ‹\u00E0› не будет совпадать с последовательностью "\u0061\u0300". Именно такое поведение проявляют другие диалекты регулярных выражений, рассматриваемые в книге.

В Java 7 можно установить параметр Pattern.UNICODE_CHARACTER_CLASS, чтобы включить режим соответствия сокращенных форм записи символьных классов символам Юникода. Подробнее об этом режиме рассказывается в разделе «Сокращения» в рецепте 2.3.

Наконец, при включенном параметре Pattern.UNIX_LINES только символ ‹\n› будет интерпретироваться как граница строки, а точка, крышка и знак доллара – нет. По умолчанию все символы разрыва строк в Юникоде интерпретируются как символы границы строки.

JavaScript

Если необходимо повторно применять регулярное выражение к той же самой строке, то есть выполнить обход всех совпадений или отыскать и заменить все совпадения, а не только первое, следует указать флаг /g, или «global» (глобальный).

XRegExp

При использовании библиотеки XRegExp следует установить флаг "g", как и при использовании стандартного диалекта JavaScript, если регу-

лярное выражение необходимо многократно применить к одной и той же строке. Кроме того, библиотека XRegExp добавляет флаг "n", превращающий все группы, кроме именованных, в несохраняющие. С этим флагом выражение «(…)» действует точно так же, как выражение «(?:…)». Если для сохранения вы всегда используете только именованные группы, добавляйте этот флаг, чтобы сделать свои регулярные выражения более эффективными без применения синтаксиса «(?:…)». Подробнее о группировке рассказывается в рецепте 2.9. Именованные группы описываются в рецепте 2.11.

PHP

Флаг /u предписывает библиотеке PCRE интерпретировать регулярное выражение и строку испытуемого текста как строки в кодировке UTF-8. Этот модификатор также позволяет использовать в регулярном выражении такие метасимволы Юникода, как «\x{FFFF}» и «\p{L}». Они подробно описаны в рецепте 2.7. Без этого модификатора PCRE интерпретирует каждый байт как отдельный символ, а метасимволы Юникода вызывают появление ошибки.

Флаг /U меняет поведение «максимальных» и «минимальных» квантификаторов на противоположное, добавляя дополнительный знак вопроса. Обычно квантификатор «.*» является максимальным, а квантификатор «.*?» — минимальным. При наличии флага /U квантификатор «.*» становится минимальным, а квантификатор «.*?» — максимальным. Мы настоятельно рекомендуем никогда не использовать этот флаг, так как он будет сбивать с толку программистов, которым позднее придется читать ваш программный код и которые пропустят дополнительный модификатор /U, уникальный для PHP. Кроме того, не путайте флаги /U и /u, если они встретятся где-нибудь в программном коде. Модификаторы режима чувствительны к регистру символов.

Perl

Если необходимо повторно применять регулярное выражение к той же самой строке (например, выполнить обход всех совпадений или отыскать и заменить все совпадения, а не только первое), следует указать флаг /g, или «global» (глобальный).

Если в регулярном выражении имеется переменная, например m/I am $name/, Perl будет компилировать регулярное выражение при каждом его использовании, потому что содержимое переменной $name может измениться. Отменить это поведение можно с помощью модификатора /o. Выражение m/I am $name/o будет компилироваться только при первом его применении, а при последующих будет задействована скомпилированная версия. Если содержимое переменной $name изменится, это никак не отразится на регулярном выражении. Информация о том, как управлять повторной компиляцией регулярных выражений, приводится в рецепте 3.3.

Если в регулярном выражении используется сокращенная форма записи символьных классов или метасимволы границ слов, можно воспользоваться одним из флагов: /d, /u, /a или /l, которые управляют соответствием сокращенных форм записи классов и метасимволов границ слов только символам ASCII или символам Юникода, если их использование предусмотрено текущими региональными настройками. Подробнее о применении этих флагов в Perl рассказывается в разделах «Варианты» рецептов 2.3 и 2.6.

Python

В языке Python имеются два дополнительных параметра, которые изменяют значение метасимвола границы слова (рецепт 2.6) и сокращенных форм записи символьных классов ‹\w›, ‹\d› и ‹\s›, а также их инвертированные версии (рецепт 2.3). По умолчанию эти метасимволы обслуживают только алфавитные символы ASCII, цифры и пробельные символы.

Параметр re.LOCALE, или re.L, обеспечивает чувствительность этих метасимволов к текущим региональным настройкам. Региональные настройки определяют, какие символы будут интерпретироваться этими метасимволами регулярных выражений как алфавитные символы, цифры и пробельные символы. Этот параметр необходимо использовать, когда испытуемый текст не является строкой Юникода и необходимо, чтобы также обрабатывались такие символы, как символы с диакритическими знаками.

Параметр re.UNICODE, или re.U, обеспечивает зависимость метасимволов от стандарта Юникод. Все символы, определяемые в Юникоде как буквы, цифры и пробельные символы, будут соответственно интерпретироваться этими метасимволами. Этот параметр необходимо использовать, когда испытуемый текст, к которому применяется регулярное выражение, является строкой Юникода.

Ruby

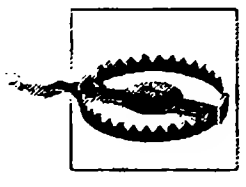
Фабричная функция Regexp.new() принимает необязательный третий аргумент, в котором указывается кодировка символов, поддерживаемая регулярным выражением. Если не указывать кодировку, будет применяться та же кодировка, которая используется в исходном файле. В большинстве случаев использование кодировки исходных файлов является правильным выбором.

Чтобы выбрать кодировку явно, необходимо передать в этом аргументе единственный символ. Аргумент не чувствителен к регистру символов. Возможные значения приводятся ниже:

- n Происходит от слова «None» (нет). Каждый байт строки будет интерпретироваться как отдельный символ. Это значение следует использовать для текста в кодировке ASCII.
- e Активизирует кодировку «EUC» для языков Дальнего Востока.

- s Активизирует японскую кодировку «Shift-JIS».
- u Активизирует кодировку UTF-8, в которой для представления одного символа может использоваться от одного до четырех байт и поддерживаются все языки, поддерживаемые стандартом Юникод (в число которых входят все живые языки независимо от их значимости).

При использовании литералов регулярных выражений кодировка может выбираться с помощью модификаторов /n, /e, /s и /u. В каждом отдельном регулярном выражении может использоваться только один из этих модификаторов. Они могут использоваться в комбинации с любыми другими модификаторами /x, /i и /m.



Не путайте модификатор /s в языке Ruby с похожим модификатором в языках Perl, Java или .NET. В Ruby модификатор /s активизирует применение кодировки «Shift-JIS». В Perl и в большинстве других диалектов регулярных выражений он включает режим «точке соответствуют границы строки». В Ruby этот режим включается с помощью флага /m.

См. также

Эффекты, производимые режимами сопоставления, подробно описываются в главе 2. Там же описывается порядок использования модификаторов режима внутри регулярного выражения.

Свободное форматирование: рецепт 2.18

Нечувствительность к регистру символов: «Поиск без учета регистра символов» в рецепте 2.1

Точке соответствуют границы строк: рецепт 2.4

Символам ^ и $ соответствуют границы строк: рецепт 2.5

В рецептах 3.1 и 3.3 описывается порядок использования литералов регулярных выражений в исходных текстах программ и создания объектов регулярных выражений. Параметры регулярных выражений устанавливаются на этапе их создания.

3.5. Проверка возможности совпадения в пределах испытуемой строки

Задача

Необходимо проверить, будет ли найдено совпадение с определенным регулярным выражением в определенной строке. Достаточно частичного совпадения. Например, регулярному выражению `‹regex•pattern›` соответствует часть строки `The regex pattern can be found`. По условиям задачи не требуется беспокоиться о каких-либо особенностях сопоставления. Достаточно просто узнать, имеется совпадение или нет.

Решение

C#

Для быстрых однократных проверок можно использовать статические функции:

```
bool foundMatch = Regex.IsMatch(subjectString, "regex pattern");
```

Если регулярное выражение вводится конечным пользователем, следует использовать статические функции с полноценной обработкой исключений:

```
bool foundMatch = false;
try {
    foundMatch = Regex.IsMatch(subjectString, UserInput);
} catch (ArgumentNullException ex) {
    // Нельзя передавать значение null в качестве регулярного выражения
    // или строки с испытуемым текстом
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

Чтобы повторно использовать то же самое выражение, необходимо создать объект Regex**:**

```
Regex regexObj = new Regex("regex pattern");
bool foundMatch = regexObj.IsMatch(subjectString);
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex **с полноценной обработкой исключений:**

```
bool foundMatch = false;
try {
    Regex regexObj = new Regex(UserInput);
    try {
        foundMatch = regexObj.IsMatch(subjectString);
    } catch (ArgumentNullException ex) {
        // Нельзя передавать значение null в качестве регулярного выражения
        // или строки с испытуемым текстом
    }
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

Для быстрых однократных проверок можно использовать статические функции:

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "regex pattern")
```

Если регулярное выражение вводится конечным пользователем, следует использовать статические функции с полноценной обработкой исключений:

```
Dim FoundMatch As Boolean
Try
    FoundMatch = Regex.IsMatch(SubjectString, UserInput)
Catch ex As ArgumentNullException
    'Нельзя передавать значение Nothing в качестве регулярного выражения
    'или строки с испытуемым текстом
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Чтобы повторно использовать то же самое выражение, необходимо создать объект Regex**:**

```
Dim RegexObj As New Regex("regex pattern")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Функции IsMatch() **обязательно должна передаваться строка** SubjectString **в качестве единственного аргумента, и вызов должен производиться относительно экземпляра класса** RegexObj**, а не как статическая функция класса** Regex**:**

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex **с полноценной обработкой исключений:**

```
Dim FoundMatch As Boolean
Try
    Dim RegexObj As New Regex(UserInput)
    Try
        FoundMatch = Regex.IsMatch(SubjectString)
    Catch ex As ArgumentNullException
        'Нельзя передавать значение Nothing в качестве регулярного выражения
        'или строки с испытуемым текстом
    End Try
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Java

Единственный способ проверить частичное совпадение заключается в том, чтобы создать объект класса Matcher**:**

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.find();
```

Если регулярное выражение вводится конечным пользователем, следует предусмотреть полноценную обработку исключений:

```
boolean foundMatch = false;
try {
    Pattern regex = Pattern.compile(UserInput);
    Matcher regexMatcher = regex.matcher(subjectString);
```

```
    foundMatch = regexMatcher.find();
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

JavaScript

```
if (/regex pattern/.test(subject)) {
    // Успешное совпадение
} else {
    // Попытка сопоставления завершилась неудачей
}
```

PHP

```
if (preg_match('/regex pattern/', $subject)) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Perl

Когда испытуемая строка хранится в специальной переменной $_:

```
if (m/regex pattern/) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Когда испытуемая строка хранится в переменной $subject:

```
if ($subject =~ m/regex pattern/) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

При использовании регулярного выражения, скомпилированного ранее:

```
$regex = qr/regex pattern/;
if ($subject =~ $regex) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
if re.search("regex pattern", subject):
    # Успешное совпадение
```

```
else:
    # Попытка сопоставления завершилась неудачей
```

Чтобы повторно использовать то же регулярное выражение, следует использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile("regex pattern")
if reobj.search(subject):
    # Успешное совпадение
else:
    # Попытка сопоставления завершилась неудачей
```

Ruby

```
if subject =~ /regex pattern/
    # Успешное совпадение
else
    # Попытка сопоставления завершилась неудачей
end
```

Следующий фрагмент делает то же самое:

```
if /regex pattern/ =~ subject
    # Успешное совпадение
else
    # Попытка сопоставления завершилась неудачей
end
```

Обсуждение

Наиболее часто регулярные выражения используются для проверки соответствия строки регулярному выражению. В большинстве языков программирования совпадения регулярного выражения хотя бы с частью строки достаточно, чтобы функция сопоставления вернула истинное значение. Функция сопоставления просматривает всю строку с испытуемым текстом, чтобы определить, совпадает ли регулярное выражение с какой-либо ее частью. Функция возвращает истинное значение, как только будет найдено первое совпадение. Ложное значение возвращается, когда был достигнут конец строки и при этом не было найдено ни одного совпадения.

Примеры программного кода в этом рецепте могут использоваться для проверки наличия в строке некоторых данных. Если требуется проверить, соответствует ли вся строка определенному шаблону (например, с целью проверки ввода пользователя), следует использовать следующий рецепт.

C# и VB.NET

Класс `Regex` предоставляет нам четыре перегруженных версии метода `IsMatch()`, два из которых являются статическими. Это позволяет вызывать метод `IsMatch()` с различными аргументами. Испытуемая строка

всегда передается в первом параметре. Это та самая строка, в которой регулярное выражение будет пытаться отыскать совпадение. Первый аргумент не может иметь значение `null`. В противном случае метод `IsMatch()` возбудит исключение `ArgumentNullException`.

Проверку можно выполнить одной строкой программного кода, вызвав статический метод `Regex.IsMatch()`, без создания объекта `Regex`. В этом случае регулярное выражение передается во втором аргументе, а параметры регулярного выражения – в необязательном третьем аргументе. Если регулярное выражение содержит синтаксическую ошибку, метод `IsMatch()` возбудит исключение `ArgumentException`. Если регулярное выражение не содержит ошибок, метод вернет значение `true` в случае, когда в строке имеется совпадение, и `false` – если совпадение не было найдено.

Если одно и то же регулярное выражение требуется применить для проверки множества строк, можно повысить производительность программного кода, предварительно создав объект `Regex` и вызывая метод `IsMatch()` этого объекта. Этот метод принимает единственный обязательный аргумент – испытуемую строку. Во втором необязательном аргументе можно указать позицию символа, начиная с которого следует выполнять поиск. Фактически число, указываемое во втором аргументе, – это количество символов от начала испытуемой строки, которые должны игнорироваться регулярным выражением. Это может быть удобно, когда часть строки уже была проверена и необходимо выполнить проверку в оставшейся части строки. Указываемое число должно быть больше или равно нулю и меньше или равно длине строки. В противном случае метод `IsMatch()` возбудит исключение `ArgumentOutOfRangeException`.

Java

Чтобы проверить наличие совпадения регулярного выражения с полной строкой или с ее частью, необходимо создать объект `Matcher`, как описывается в рецепте 3.3. После этого можно вызвать метод `find()` вновь созданного или повторно инициализированного объекта `Matcher`.

Методы `String.matches()`, `Pattern.matches()` или `Matcher.matches()` непригодны для решения поставленной здесь задачи, так как требуют, чтобы регулярное выражение совпадало с полной строкой.

JavaScript

Чтобы проверить наличие совпадения регулярного выражения с частью строки, необходимо вызвать метод `test()` регулярного выражения. Испытуемая строка передается этому методу в виде единственного аргумента.

Метод `regexp.test()` возвращает `true`, если регулярному выражению соответствует вся испытуемая строка или ее часть, и `false` – в противном случае.

PHP

Функция `preg_match()` может использоваться для разных целей. В простейшем случае ей передаются всего два обязательных аргумента: строка с регулярным выражением и строка с испытуемым текстом, который необходимо проверить с помощью регулярного выражения. Если совпадение было найдено, функция `preg_match()` вернет значение `1`; если в испытуемом тексте не было обнаружено ни одного совпадения, она вернет значение `0`.

Необязательные аргументы, которые могут передаваться функции `preg_match()`, описываются в последующих рецептах.

Perl

В языке Perl синтаксическая конструкция `m//` фактически является не просто контейнером для хранения регулярного выражения, а оператором регулярного выражения. При использовании оператора `m//` самого по себе в качестве испытуемого текста используется содержимое переменной `$_`.

Если необходимо применить оператор сопоставления `m//` к содержимому другой переменной, следует использовать оператор связывания `=~`, который свяжет оператор регулярного выражения с указанной переменной. Оператор сопоставления с шаблоном вернет значение `true`, если регулярному выражению соответствует часть испытуемой строки, и `false` — если ни одного совпадения не было найдено.

Если необходимо убедиться в отсутствии совпадений с регулярным выражением в строке, можно использовать оператор `!~`, который является инвертированной версией оператора `=~`.

Python

В модуле `re` имеется функция `search()`, которая выполняет поиск совпадений частей строки с регулярным выражением. Регулярное выражение передается функции в первом аргументе, а испытуемая строка — во втором. Параметры регулярного выражения можно передать в третьем необязательном аргументе.

Функция `re.search()` вызывает функцию `re.compile()` и затем вызывает метод `search()` объекта скомпилированного регулярного выражения. Этот метод принимает единственный аргумент — испытуемую строку.

Если регулярное выражение обнаружит совпадение, функция `search()` вернет экземпляр класса `MatchObject`. Если совпадение не будет найдено, функция `search()` вернет значение `None`. При использовании возвращаемого значения в условном операторе `if` объект `MatchObject` оценивается как значение `True`, а значение `None` оценивается как `False`. В последующих рецептах этой главы демонстрируется, как использовать информацию, хранящуюся в объекте `MatchObject`.

Не следует путать функции `search()` **и** `match()`**. Функция** `match()` **не может использоваться для поиска совпадения в середине строки. Функция** `match()` **используется в следующем рецепте.**

Ruby

Оператор `=~` – это оператор сопоставления с шаблоном. Чтобы отыскать в строке первое совпадение с регулярным выражением, необходимо поместить его между регулярным выражением и строкой. Оператор возвращает номер позиции найденного совпадения. Если совпадение не будет обнаружено, оператор вернет значение `nil`.

Этот оператор реализован в обоих классах, `Regexp` и `String`. В версии Ruby 1.8 не имеет никакого значения, экземпляр какого класса будет располагаться слева или справа от оператора. В версии Ruby 1.9 с этим связан побочный эффект, имеющий отношение к именованным сохраняющим группам. Подробнее об этом рассказывается в рецепте 3.9.

Во всех других фрагментах программного кода на языке Ruby в этой книге мы будем помещать испытуемую строку слева от оператора `=~`, а регулярное выражение – справа. Это соответствует синтаксису языка Perl, из которого Ruby заимствовал оператор `=~`, и позволяет обойти особенности версии Ruby 1.9, связанные с именованными сохраняющими группами, которые для многих могут оказаться неожиданными.

См. также

Рецепт 3.6, где демонстрируется код, проверяющий совпадение регулярного выражения со всей испытуемой строкой.

Рецепт 3.7, где показано, как получить фрагмент, фактически совпавший с регулярным выражением.

3.6. Проверка совпадения со всей испытуемой строкой

Задача

Необходимо проверить, соответствует ли строка целиком некоторому регулярному выражению. То есть необходимо убедиться, что регулярное выражение содержит шаблон, которому соответствует строка целиком, от начала до конца. Например, допустим, что имеется регулярное выражение `‹regex•pattern›`, которое будет совпадать со входным текстом, состоящим из строки `regex pattern`, но не с более длинной строкой `The regex pattern can be found`.

Решение

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
bool foundMatch = Regex.IsMatch(subjectString, @"\Aregex pattern\Z");
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект Regex:

```
Regex regexObj = new Regex(@"\Aregex pattern\Z");
bool foundMatch = regexObj.IsMatch(subjectString);
```

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "\Aregex pattern\Z")
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект Regex:

```
Dim RegexObj As New Regex("\Aregex pattern\Z")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Функции IsMatch() обязательно должна передаваться строка SubjectString в качестве единственного аргумента, и вызов должен производиться относительно экземпляра класса RegexObj, а не как статическая функция класса Regex:

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Java

Для однократной проверки можно использовать статическую функцию:

```
boolean foundMatch = subjectString.matches("regex pattern");
```

В случае многократного использования того же самого регулярного выражения предпочтительнее скомпилировать регулярное выражение и создать объект совпадения:

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.matches(subjectString);
```

JavaScript

```
if (/^regex pattern$/.test(subject)) {
    // Успешное совпадение
} else {
    // Попытка сопоставления завершилась неудачей
}
```

PHP

```
if (preg_match('/\Aregex pattern\Z/', $subject)) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Perl

```
if ($subject =~ m/\Aregex pattern\Z/) {
    # Успешное совпадение
} else {
    # Попытка сопоставления завершилась неудачей
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
if re.match(r"regex pattern\Z", subject):
    # Успешное совпадение
else:
    # Попытка сопоставления завершилась неудачей
```

В случае многократного использования того же самого регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"regex pattern\Z")
if reobj.match(subject):
    # Успешное совпадение
else:
    # Попытка сопоставления завершилась неудачей
```

Ruby

```
if subject =~ /\Aregex pattern\Z/
    # Успешное совпадение
else
    # Попытка сопоставления завершилась неудачей
End
```

Обсуждение

Обычно успех сопоставления с регулярным выражением говорит о том, что *где-то* внутри испытуемого текста имеется совпадение с шаблоном. Во многих ситуациях бывает необходимо убедиться, что регулярному выражению соответствует не какая-то часть, а *весь* испытуемый текст целиком. Пожалуй, наиболее типичная ситуация, когда необходимо выяснить, совпадает ли текст целиком, – это проверка ввода. Если пользователь вводит номер телефона или IP-адрес и включает посторонние символы, необходимо отклонить его ввод.

Решения, где используются якорные метасимволы ‹$› и ‹\Z›, также могут применяться при построчной обработке файлов (рецепт 3.21), когда используемый механизм извлечения строк оставляет символы завершения строки в конце строки. Как описывается в рецепте 2.5, эти якорные метасимволы также совпадают с последним символом завершения строки, фактически позволяя игнорировать его.

В последующих подразделах подробно описываются решения для различных языков программирования.

C# и VB.NET

Класс `Regex` в платформе .NET Framework не имеет функции, позволяющей убедиться, что строка целиком соответствует регулярному выражению. Решение заключается в том, чтобы в начало регулярного выражения добавить якорный метасимвол ‹\A›, совпадающий с началом строки, а в конец выражения – якорный метасимвол ‹\Z›, совпадающий с концом строки. Благодаря этому строка может совпадать с регулярным выражением только целиком, в противном случае совпадения вообще не будет. Если в регулярном выражении используется конструкция выбора, такая как ‹one|two|three›, ее необходимо заключить в группу перед добавлением якорных метасимволов: ‹\A(?:one|two|three)\Z›.

После внесения исправлений в регулярное выражение, которые обеспечат совпадение только с целой строкой, можно использовать метод `IsMatch()`, который был описан в предыдущем рецепте.

Java

В языке Java имеются три метода с именем `matches()`. Все они проверяют, совпадает ли регулярное выражение со строкой целиком. Эти методы обеспечивают быстрый способ проверки ввода без заключения регулярного выражения в якорные метасимволы, совпадающие с началом и с концом строки.

Метод `matches()` класса `String` принимает регулярное выражение в виде единственного аргумента. Он возвращает значение `true` или `false` как признак того, совпадает ли регулярное выражение со всей строкой целиком. Метод `matches()` класса `Pattern` принимает две строки: первая – это регулярное выражение, а вторая – испытуемая строка. В качестве испытуемой строки методу `Pattern.matches()` в действительности можно передавать любой объект типа `CharSequence`. Это единственная причина отказа от использования `String.matches()` в пользу `Pattern.matches()`.

Оба метода, `String.matches()` и `Pattern.matches()`, при каждом обращении производят компиляцию регулярного выражения, вызывая `Pattern.compile("regex").matcher(subjectString).matches()`. Так как регулярное выражение компилируется каждый раз, эти методы желательно применять только при однократном использовании регулярного выражения (например, для проверки одного поля ввода в форме) или когда не требуется высокая производительность. Эти методы не предоставляют воз-

можность указывать параметры сопоставления за пределами регулярного выражения. Если регулярное выражение содержит синтаксическую ошибку, возбуждается исключение `PatternSyntaxException`.

В случае многократного использования одного и того же регулярного выражения для повышения эффективности необходимо скомпилировать его и создать объект `Matcher`, как описывается в рецепте 3.3. После этого можно будет вызывать метод `matches()` созданного экземпляра класса `Matcher`. Этот метод не принимает никаких аргументов, потому что испытуемая строка указывается при создании объекта сопоставления.

JavaScript

В языке JavaScript отсутствует функция, позволяющая убедиться, что строка целиком соответствует регулярному выражению. Решение заключается в том, чтобы в начало регулярного выражения добавить якорный метасимвол ‹^›, а в конец выражения – якорный метасимвол ‹$›. При этом не следует устанавливать флаг `/m` регулярного выражения. Только в отсутствие флага `/m` символ крышки и знак доллара будут совпадать исключительно с началом и с концом испытуемой строки. При установленном флаге `/m` они также будут совпадать с границами строк в середине испытуемого текста.

После добавления якорных метасимволов в регулярное выражение его можно использовать с методом `regexp.test()`, описанным в предыдущем рецепте.

PHP

В языке PHP отсутствует функция, позволяющая убедиться, что строка целиком соответствует регулярному выражению. Решение заключается в том, чтобы в начало регулярного выражения добавить якорный метасимвол ‹\A›, совпадающий с началом строки, а в конец выражения – якорный метасимвол ‹\Z›, совпадающий с концом строки. Благодаря этому строка может совпадать с регулярным выражением только целиком, в противном случае совпадения вообще не будет. Если в регулярном выражении используется конструкция выбора, такая как ‹one|two|three›, ее необходимо заключить в группу перед добавлением якорных метасимволов: ‹\A(?:one|two|three)\Z›.

После внесения исправлений в регулярное выражение, которые обеспечат совпадение только с целой строкой, можно использовать функцию `preg_match()`, описанную в предыдущем рецепте.

Perl

В языке Perl имеется всего один оператор сопоставления с шаблоном, который удовлетворяется совпадением с частью строки. Если необходимо проверить, соответствует ли регулярному выражению вся строка целиком, нужно в начало регулярного выражения добавить якорный ме-

тасимвол ‹\A›, совпадающий с началом строки, а в конец выражения – якорный метасимвол ‹\Z›, совпадающий с концом строки. Благодаря этому строка может совпадать с регулярным выражением только целиком, в противном случае совпадения вообще не будет. Если в регулярном выражении используется конструкция выбора, такая как ‹one|two|three›, ее необходимо заключить в группу перед добавлением якорных метасимволов: ‹\A(?:one|two|three)\Z›.

После внесения исправлений в регулярное выражение, которые обеспечат совпадение только с целой строкой, его можно использовать, как описано в предыдущем рецепте.

Python

Функция `match()` очень схожа с функцией `search()`, описанной в предыдущем рецепте. Основное отличие состоит в том, что функция `match()` требует, чтобы регулярное выражение совпадало исключительно с начала испытуемой строки. Если регулярное выражение не совпадает с началом строки, функция `match()` тут же возвращает значение `None`. Функция `search()`, в свою очередь, будет пытаться отыскать совпадение с регулярным выражением в каждой последующей позиции в строке, пока либо не обнаружит совпадение, либо не достигнет конца испытуемой строки.

Функция `match()` не требует, чтобы вся строка целиком соответствовала регулярному выражению. Она допускает совпадение с частью строки при условии, что это совпадение начинается в начале строки. Если необходимо проверить, соответствует ли регулярному выражению вся строка целиком, нужно в конец выражения добавить якорный метасимвол ‹\Z›, совпадающий с концом строки.

Ruby

Класс `Regex` в языке Ruby не имеет функции, позволяющей убедиться, что строка целиком соответствует регулярному выражению. Для решения поставленной задачи необходимо в начало регулярного выражения добавить якорный метасимвол ‹\A›, совпадающий с началом строки, а в конец выражения – якорный метасимвол ‹\Z›, совпадающий с концом строки. Благодаря этому строка может совпадать с регулярным выражением только целиком, в противном случае совпадения вообще не будет. Если в регулярном выражении используется конструкция выбора, такая как ‹one|two|three›, ее необходимо заключить в группу перед добавлением якорных метасимволов: ‹\A(?:one|two|three)\Z›.

После внесения исправлений в регулярное выражение, которые обеспечат совпадение только с целой строкой, можно использовать оператор =~, описанный в предыдущем рецепте.

См. также

Рецепт 2.5, подробно рассказывающий о том, как работают якорные метасимволы.

Рецепты 2.8 и 2.9, где описываются оператор выбора и группировка. Если регулярное выражение использует конструкцию выбора за пределами какой-либо группы, ее необходимо заключить в группу перед добавлением якорных метасимволов. Если регулярное выражение не использует конструкцию выбора или использует ее только в пределах групп, для обеспечения корректной работы якорей дополнительная группировка не требуется.

Рецепт 3.7 может пригодиться, если допустимо совпадение с регулярным выражением части строки.

3.7. Извлечение текста совпадения

Задача

Имеется регулярное выражение, совпадающее с частью испытуемого текста. Необходимо извлечь совпавшие фрагменты текста. Если для регулярного выражения в строке имеется более одного совпадения, необходимо извлечь только первое из них. Например, в результате применения регулярного выражения `\d+` к строке `Do you like 13 or 42?` должно быть извлечено совпадение `13`.

Решение

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
string resultString = Regex.Match(subjectString, @"\d+").Value;
```

Если регулярное выражение вводится конечным пользователем, следует использовать статические функции с полноценной обработкой исключений:

```
string resultString = null;
try {
    resultString = Regex.Match(subjectString, @"\d+").Value;
} catch (ArgumentNullException ex) {
    // Нельзя передавать значение null в качестве регулярного выражения
    // или строки с испытуемым текстом
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

Чтобы повторно использовать то же самое выражение, необходимо создать объект `Regex`:

```
Regex regexObj = new Regex(@"\d+");
string resultString = regexObj.Match(subjectString).Value;
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex **с полноценной обработкой исключений:**

```
string resultString = null;
try {
    Regex regexObj = new Regex(@"\d+");
    try {
        resultString = regexObj.Match(subjectString).Value;
    } catch (ArgumentNullException ex) {
        // Нельзя передавать значение null в качестве
        // строки с испытуемым текстом
    }
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim ResultString = Regex.Match(SubjectString, "\d+").Value
```

Если регулярное выражение вводится конечным пользователем, следует использовать статические функции с полноценной обработкой исключений:

```
Dim ResultString As String = Nothing
Try
    ResultString = Regex.Match(SubjectString, "\d+").Value
Catch ex As ArgumentNullException
    'Нельзя передавать значение Nothing в качестве регулярного выражения
    'или строки с испытуемым текстом
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Чтобы повторно использовать то же самое выражение, необходимо создать объект Regex**:**

```
Dim RegexObj As New Regex("\d+")
Dim ResultString = RegexObj.Match(SubjectString).Value
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex **с полноценной обработкой исключений:**

```
Dim ResultString As String = Nothing
Try
    Dim RegexObj As New Regex("\d+")
    Try
        ResultString = RegexObj.Match(SubjectString).Value
    Catch ex As ArgumentNullException
```

```
        'Нельзя передавать значение Nothing в качестве
        'строки с испытуемым текстом
    End Try
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Java

Чтобы отыскать и сохранить совпадение, необходимо создать объект Matcher:

```
String resultString = null;
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    resultString = regexMatcher.group();
}
```

Если регулярное выражение вводится конечным пользователем, следует предусматривать полноценную обработку исключений:

```
String resultString = null;
try {
    Pattern regex = Pattern.compile("\\d+");
    Matcher regexMatcher = regex.matcher(subjectString);
    if (regexMatcher.find()) {
        resultString = regexMatcher.group();
    }
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

JavaScript

```
var result = subject.match(/\d+/);
if (result) {
    result = result[0];
} else {
    result = '';
}
```

PHP

```
if (preg_match('/\d+/', $subject, $groups)) {
    $result = $groups[0];
} else {
    $result = '';
}
```

Perl

```
if ($subject =~ m/\d+/) {
    $result = $&;
```

```
} else {
    $result = '';
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
matchobj = re.search("regex pattern", subject)
if matchobj:
    result = matchobj.group()
else:
    result = ""
```

Чтобы иметь возможность использовать то же самое регулярное выражение повторно, следует создать объект скомпилированного регулярного выражения:

```
reobj = re.compile("regex pattern")
matchobj = reobj.search(subject)
if match:
    result = matchobj.group()
else:
    result = ""
```

Ruby

Можно использовать оператор =~ и его специальную переменную $&:

```
if subject =~ /regex pattern/
    result = $&
else
    result = ""
end
```

Как вариант можно задействовать метод `match()` объекта `Regexp`:

```
matchobj = /regex pattern/.match(subject)
if matchobj
    result = matchobj[0]
else
    result = ""
end
```

Обсуждение

Извлечение фрагмента строки, соответствующего шаблону, – это еще одна из основных задач, решаемых с помощью регулярных выражений. Во всех языках программирования, рассматриваемых в этой книге, имеется простой способ получения первого совпадения с регулярным выражением в строке. Функция начнет попытки применения регулярного выражения с начала строки и продолжит просматривать ее, пока не будет обнаружено совпадение.

.NET

В классе Regex платформы .NET отсутствует функция-член, которая возвращала бы фрагмент, совпавший с регулярным выражением. Но в нем имеется метод Match(), возвращающий экземпляр класса Match. Объекты класса Match имеют свойство Value, в котором хранится текст, совпавший с регулярным выражением. Если попытка найти совпадения с регулярным выражением потерпела неудачу, этот метод все равно вернет объект Match, свойство Value которого будет содержать пустую строку.

Всего имеется пять перегруженных версий метода Match(). В первом аргументе всегда передается строка с испытуемым текстом, в котором требуется отыскать совпадения с регулярным выражением. Этот аргумент не может иметь значение null. В противном случае метод Match() возбудит исключение ArgumentNullException.

Если регулярное выражение необходимо использовать небольшое число раз, можно воспользоваться статическим методом. В этом случае вторым аргументом методу передается регулярное выражение. В третьем необязательном аргументе можно передать параметры регулярного выражения. Если регулярное выражение содержит синтаксическую ошибку, будет возбуждено исключение ArgumentException.

Если одно и то же регулярное выражение требуется применить для проверки множества строк, можно повысить производительность программного кода, предварительно создав объект Regex и затем вызывая метод Match() этого объекта. Этот метод принимает единственный обязательный аргумент – испытуемую строку. Во втором необязательном аргументе можно указать позицию символа, начиная с которого следует выполнять поиск. Фактически число, указываемое во втором аргументе, – это количество символов от начала испытуемой строки, которые должны игнорироваться регулярным выражением. Это удобно, если часть строки уже была проверена и необходимо выполнить проверку в оставшейся части строки. Указываемое число должно быть в диапазоне от нуля до величины длины строки. В противном случае метод Match() возбудит исключение ArgumentOutOfRangeException.

Если методу передается второй аргумент с начальной позицией, то можно также передать третий аргумент – длину подстроки, в пределах которой следует искать совпадения с регулярным выражением. Это число должно быть больше или равно нулю и не превышать длины испытуемой строки (первый аргумент) минус начальное смещение (второй аргумент). Например, вызов regexObj.Match("123456", 3, 2) попытается отыскать совпадение в подстроке "45". Если величина третьего аргумента будет больше, чем длина испытуемой строки, метод Match() возбудит исключение ArgumentOutOfRangeException. Если величина третьего аргумента не будет превышать длину испытуемой строки, но сумма второго и третьего аргументов окажется больше длины строки, то будет возбуждено другое исключение: IndexOutOfRangeException. Если вы позволяете конечному пользователю указывать начальную и конечную позиции,

то либо проверяйте полученные значения перед вызовом метода `Match()`, либо предусмотрите обработку обоих типов исключений.

Статические методы не имеют аргументов, с помощью которых можно было бы указать, в какой части строки следует выполнять поиск совпадения с регулярным выражением.

Java

Чтобы получить часть строки, совпавшей с регулярным выражением, необходимо создать объект класса `Matcher`, как описывается в рецепте 3.3, и затем вызвать метод `find()` этого объекта без аргументов. Если метод `find()` вернет значение `true`, следует вызвать метод `group()` без аргументов, чтобы извлечь текст, совпавший с регулярным выражением. Если метод `find()` вернет значение `false`, метод `group()` вызывать не следует, так как в этом случае будет возбуждено исключение `IllegalStateException`.

Метод `Matcher.find()` принимает один необязательный аргумент с начальной позицией в испытуемой строке. Этот аргумент можно использовать, чтобы организовать поиск, начиная с определенной позиции в строке. Если указать значение ноль, поиск начнется с начала строки. Если указать значение меньше нуля или больше длины испытуемой строки, будет возбуждено исключение `IndexOutOfBoundsException`.

Если опустить аргумент, функция `find()` всякий раз будет начинать поиск с позиции, следующей за предыдущим совпадением, найденным ею. Если вызвать функцию `find()` сразу после вызова метода `Pattern.matcher()` или `Matcher.reset()`, она начнет поиск с начала строки.

JavaScript

Метод `string.match()` принимает единственный аргумент с регулярным выражением. Регулярное выражение можно передавать в виде литерала регулярного выражения, в виде объекта регулярного выражения или в виде строки. Если регулярное выражение передается в виде строки, метод `string.match()` создаст временный объект `regexp`.

Если попытка сопоставления потерпит неудачу, метод `string.match()` вернет значение `null`. Это позволяет отличать случаи, когда в строке отсутствуют совпадения с регулярным выражением и когда для регулярного выражения обнаруживается совпадение с нулевой длиной. Это означает, что невозможно будет напрямую отобразить результат, так как на экране может появиться текст «null» или сообщение об ошибке обращения к несуществующему объекту.

Если попытка сопоставления увенчается успехом, метод `string.match()` вернет массив с полной информацией о совпадении. Нулевой элемент массива – это строка с текстом, совпавшим с регулярным выражением.

Регулярное выражение не должно содержать флаг `/g`. В противном случае поведение метода `string.match()` будет отличаться от описанного здесь, как объясняется в рецепте 3.10.

PHP

Функция `preg_match()`, рассматривавшаяся в двух предыдущих рецептах, принимает необязательный третий аргумент, в котором будет сохраняться текст, совпавший с регулярным выражением и с его сохраняющими группами. Если функция `preg_match()` возвращает значение `1`, в указанной переменной будет храниться массив строк. В нулевом элементе массива хранится совпадение со всем регулярным выражением. Остальные элементы массива будут описаны в рецепте 3.9.

Perl

Когда оператор `m//` сопоставления с шаблоном обнаруживает совпадение, он записывает значения в несколько специальных переменных. Одна из таких переменных, `$&`, хранит часть строки, совпавшей с регулярным выражением. Другие специальные переменные будут описаны в последующих рецептах.

Python

В рецепте 3.5 описывается функция `search()`. В этом рецепте экземпляр класса `MatchObject`, возвращаемый функцией `search()`, сохранялся в переменной. Чтобы получить часть строки, совпавшей с регулярным выражением, следует вызвать метод `group()` объекта совпадения без аргументов.

Ruby

В рецепте 3.8 описываются переменная `$~` и объект `MatchData`. В строковом контексте этот объект преобразуется в текст, совпавший с регулярным выражением. В контексте массива этот объект преобразуется в массив, нулевой элемент которого хранит совпадение со всем регулярным выражением.

`$&` – это специальная переменная, доступная только для чтения. Она представляет собой псевдоним для `$~[0]`, где хранится совпадение со всем регулярным выражением.

См. также

Рецепт 3.5, где демонстрируется код, проверяющий наличие совпадения с регулярным выражением в пределах испытуемой строки без извлечения фактического совпадения.

Рецепт 3.8, где демонстрируется код, определяющий позицию и длину совпадения.

Рецепт 3.9, где демонстрируется код, извлекающий фрагмент испытуемой строки, совпавший с определенной частью регулярного выражения (сохраняющей группой).

Рецепт 3.10, где демонстрируется код, извлекающий список всех совпадений, которые были найдены регулярным выражением в испытуемой строке.

Рецепт 3.11, где демонстрируется код, выполняющий обход всех совпадений, которые были найдены регулярным выражением в испытуемой строке.

3.8. Определение позиции и длины совпадения

Задача

Вместо извлечения подстроки, совпавшей с регулярным выражением, как показано в предыдущем рецепте, необходимо определить начальную позицию и длину совпадения. При наличии этой информации можно будет извлечь совпадение другими средствами или выполнить обработку части оригинальной строки, совпавшей с регулярным выражением.

Решение

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
int matchstart, matchlength = -1;
Match matchResult = Regex.Match(subjectString, @"\d+");
if (matchResult.Success) {
    matchstart = matchResult.Index;
    matchlength = matchResult.Length;
}
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект Regex**:**

```
int matchstart, matchlength = -1;
Regex regexObj = new Regex(@"\d+");
Match matchResult = regexObj.Match(subjectString).Value;
if (matchResult.Success) {
    matchstart = matchResult.Index;
    matchlength = matchResult.Length;
}
```

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim MatchStart = -1
Dim MatchLength = -1
Dim MatchResult = Regex.Match(SubjectString, "\d+")
If MatchResult.Success Then
```

```
    MatchStart = MatchResult.Index
    MatchLength = MatchResult.Length
End If
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект Regex**:**

```
Dim MatchStart = -1
Dim MatchLength = -1
Dim RegexObj As New Regex("\d+")
Dim MatchResult = RegexObj.Match(SubjectString, "\d+")
If MatchResult.Success Then
    MatchStart = MatchResult.Index
    MatchLength = MatchResult.Length
End If
```

Java

```
int matchStart, matchLength = -1;
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    matchStart = regexMatcher.start();
    matchLength = regexMatcher.end() - matchStart;
}
```

JavaScript

```
var matchstart = -1;
var matchlength = -1;
var match = /\d+/.exec(subject);
if (match) {
    matchstart = match.index;
    matchlength = match[0].length;
}
```

PHP

```
if (preg_match('/\d+/', $subject, $groups, PREG_OFFSET_CAPTURE)) {
    $matchstart = $groups[0][1];
    $matchlength = strlen($groups[0][0]);
}
```

Perl

```
if ($subject =~ m/\d+/g) {
    $matchstart = $-[0];
    $matchlength = $+[0] - $-[0];
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
matchobj = re.search(r"\d+", subject)
if matchobj:
    matchstart = matchobj.start()
    matchlength = matchobj.end() - matchstart
```

В случае многократного использования того же самого регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"\d+")
matchobj = reobj.search(subject)
if matchobj:
    matchstart = matchobj.start()
    matchlength = matchobj.end() - matchstart
```

Ruby

Можно использовать оператор =~ и его специальную переменную $~:

```
if subject =~ /regex pattern/
    matchstart = $~.begin()
    matchlength = $~.end() - matchstart
end
```

Как вариант можно задействовать метод `match()` **объекта** `Regexp`**:**

```
matchobj = /regex pattern/.match(subject)
if matchobj
    matchstart = matchobj.begin()
    matchlength = matchobj.end() - matchstart
end
```

Обсуждение

.NET

Для получения позиции и длины совпадения в решении использован тот же метод `Regex.Match()`**, что был описан в предыдущем рецепте. На этот раз использовались свойства** `Index` **и** `Length` **объекта** `Match`**, возвращаемого методом** `Regex.Match()`**.**

Свойство `Index` **хранит номер позиции в испытуемой строке, где начинается совпадение с регулярным выражением. Если совпадение начинается с начала строки, значением свойства** `Index` **будет ноль. Если совпадение начинается со второго символа в строке, значением свойства** `Index` **будет единица. Максимально возможное значение свойства** `Index` **равно длине строки. Такое может произойти, когда для регулярного выражения обнаруживается совпадение нулевой длины в конце строки. Например, регулярное выражение, состоящее только из якорного метасимвола** `\Z`**, всегда совпадает только в конце строки.**

Свойство `Length` **хранит число совпавших символов. Вполне допустимым считается, когда совпадение имеет нулевую длину. Например, ре-**

гулярное выражение, состоящее только из метасимвола границы слова `\b`, будет обнаруживать совпадение нулевой длины в начале первого слова в строке.

Если попытка сопоставления не увенчалась успехом, метод `Regex.Match()` все равно вернет объект `Match`, оба свойства `Index` и `Length` которого будут равны нулю. Эти значения могут также появляться и в случае успеха сопоставления. Регулярное выражение, состоящее только из якорного метасимвола начала строки `\A`, будет обнаруживать совпадение нулевой длины в начале строки. Вследствие этого нельзя полагаться на свойства `Match.Index` и `Match.Length` в определении успешности попытки сопоставления. Вместо этого лучше использовать свойство `Match.Success`.

Java

Чтобы определить позицию и длину совпадения, следует вызвать метод `Matcher.find()`, как было описано в предыдущем рецепте. Если метод `find()` вернет значение `true`, можно вызвать метод `Matcher.start()` без аргументов, чтобы получить индекс первого символа совпадения с регулярным выражением. При вызове метода `end()` без аргументов возвращается индекс первого символа, следующего за совпадением. Чтобы определить длину, достаточно вычесть индекс первого символа совпадения из индекса первого символа, следующего за совпадением, которая может оказаться равной нулю. Если вызвать метод `start()` или `end()`, не вызвав перед этим метод `find()`, будет возбуждено исключение `IllegalStateException`.

JavaScript

Вызов метода `exec()` объекта `regexp` возвращает массив с полной информацией о совпадении. Этот массив обладает рядом дополнительных свойств. Свойство `index` хранит позицию в испытуемой строке, с которой начинается совпадение с регулярным выражением. Если совпадение начинается с начала строки, свойство `index` будет иметь значение ноль. Нулевой элемент массива хранит строку совпадения со всем регулярным выражением. Чтобы определить длину этой строки, можно воспользоваться ее свойством `length`.

Если для регулярного выражения нет ни одного совпадения в строке, метод `regexp.exec()` вернет значение `null`.

Не следует использовать свойство `lastIndex` массива, возвращаемого методом `exec()`, для определения позиции последнего символа в совпадении. В строгой реализации JavaScript свойство `lastIndex` вообще отсутствует в возвращаемом массиве, оно имеется только у самого объекта `regexp`. Но и свойство `regexp.lastIndex` тоже не следует использовать. Оно – ненадежный источник информации из-за различий между браузерами (подробнее об этом рассказывается в рецепте 3.11). Вместо этого для определения позиции, где закончилось совпадение, нужно просто сложить значения свойств `match.index` и `match[0].length`.

PHP

В предыдущем рецепте описывалось, как получить текст, совпавший с регулярным выражением, передавая третий аргумент функции `preg_match()`. Позицию совпадения можно получить, передав в четвертом аргументе значение `PREG_OFFSET_CAPTURE`. Этот аргумент определяет, что именно будет сохранять функция `preg_match()` в третьем аргументе, когда она возвращает значение `1`.

Если четвертый аргумент опущен или в нем передается нулевое значение, переменная, передаваемая в третьем аргументе, принимает массив строк. Когда в четвертом аргументе передается значение `PREG_OFFSET_CAPTURE`, переменная принимает массив массивов. Нулевой элемент массива все так же хранит общее совпадение (смотрите предыдущий рецепт), а первый и последующие элементы, как и прежде, хранят содержимое сохраняющих групп с номерами один и выше (смотрите следующий рецепт). Но вместо простой строки совпадения с регулярным выражением или сохраняющей группой элементы массива хранят массивы из двух значений: текст совпадения и позицию в строке, где это совпадение было обнаружено.

Так, для общего совпадения нулевой подэлемент нулевого элемента хранит текст, совпавший с регулярным выражением. Если этот текст передать функции `strlen()`, можно узнать его длину. Подэлемент с индексом 1 нулевого элемента хранит целое число, определяющее позицию в испытуемой строке, где начинается совпадение.

Perl

В языке Perl все начальные позиции совпадений с сохраняющими группами запоминаются в массиве `@-`, а конечные позиции – в массиве `@+`. Совпадение со всем выражением целиком сохраняется в группе с индексом `0`. Определить позицию начала совпадения со всем выражением можно из элемента массива `@-[0]`, а конца совпадения – из элемента `@+[0]`.

Python

Метод `start()` объекта `MatchObject` возвращает позицию в строке, где начинается совпадение с регулярным выражением. Метод `end()` возвращает позицию первого символа в строке, следующего за совпадением. Оба метода возвращают одно и то же значение в случае, когда обнаружено совпадение нулевой длины.

Методы `start()` и `end()` могут принимать аргумент и возвращать фрагмент текста, совпавшего с одной из сохраняющих групп в регулярном выражении. Например, вызов `start(1)` вернет позицию начала совпадения для первой сохраняющей группы, вызов `end(2)` – позицию конца совпадения для второй группы и т. д. Язык Python поддерживает до 99 сохраняющих групп. Группе с номером `0` соответствует совпадение со всем регулярным выражением. Передача методам `start()` и `end()` любого

числа, за исключением нуля, превышающего число сохраняющих групп в регулярном выражении (которых может быть до 99), вызывает исключение IndexError. Если передан номер группы, существующей в регулярном выражении, но эта группа не участвовала в совпадении, методы start() и end() вернут для этой группы значение -1.

При желании можно сохранить начальную и конечную позиции в кортеже, вызвав метод span() объекта совпадения.

Ruby

В рецепте 3.5 для поиска первого совпадения с регулярным выражением использовался оператор =~. Попутно этот оператор записывает в специальную переменную $~ экземпляр класса MatchData. Эта переменная является локальной для потоков выполнения и локальной для методов. Это означает, что содержимое этой переменной может использоваться только в пределах метода и пока в этом методе снова не будет использован оператор =~. При этом можно не беспокоиться, что содержимое переменной будет затерто в каком-нибудь другом потоке выполнения или каким-нибудь другим методом в этом потоке выполнения.

При желании можно сохранять информацию о нескольких совпадениях с регулярными выражениями, вызывая метод match() объекта Regexp. Этот метод принимает испытуемую строку в виде единственного аргумента. Метод возвращает экземпляр класса MatchData, если совпадение было найдено, или значение nil – в противном случае. Кроме того, он сохраняет тот же самый экземпляр MatchData в специальной переменной $~, но не затирает другие экземпляры класса MatchData, хранящиеся в других переменных.

Объект класса MatchData содержит полную информацию о совпадении с регулярным выражением. В рецептах 3.7 и 3.9 описывается, как извлечь текст, совпавший с регулярным выражением и с сохраняющими группами.

Метод begin() возвращает позицию в испытуемой строке, где начинается совпадение с регулярным выражением. Метод end() возвращает позицию первого символа в испытуемой строке, следующего за совпадением. Метод offset() возвращает массив начальных и конечных позиций. Все три метода принимают единственный аргумент. Чтобы получить позиции, относящиеся к совпадению со всем регулярным выражением, следует передать значение 0. Если передается положительное число, возвращаются позиции, относящиеся к совпадению с указанной сохраняющей группой. Например, вызов begin(1) вернет начало совпадения с первой сохраняющей группой.

Не следует использовать методы length() или size(), чтобы получить длину совпадения. Оба эти метода возвращают число элементов массива, в который преобразуется объект класса MatchData в контексте, когда ожидается массив, как будет описано в рецепте 3.9.

См. также

Рецепт 3.5, где демонстрируется код, проверяющий наличие совпадения с регулярным выражением в пределах испытуемой строки без извлечения фактического совпадения.

Рецепт 3.7, где показано, как получить фрагмент, фактически совпавший с регулярным выражением.

Рецепт 3.9, где демонстрируется код, извлекающий фрагмент испытуемой строки, совпавший с определенной частью регулярного выражения (сохраняющей группой).

3.9. Извлечение части совпавшего текста

Задача

Как и в рецепте 3.7, имеется регулярное выражение, совпадающее с подстрокой в испытуемом тексте, но на этот раз необходимо извлечь только одну часть совпадения из этой подстроки. Чтобы отделить части совпадений друг от друга, в регулярном выражении используется сохраняющая группировка, как описано в рецепте 2.9.

Например, в строке `Please visit http://www.regexcookbook.com for more information` регулярному выражению `‹http://([a-z0-9.-]+)›` соответствует фрагмент `http://www.regexcookbook.com`. Части регулярного выражения, заключенной в первую сохраняющую группу, соответствует фрагмент `www.regexcookbook.com`. Необходимо извлечь это доменное имя, сохраненное первой сохраняющей группой, в строковую переменную.

Для иллюстрации принципов работы с сохраняющими группами мы будем использовать это простое регулярное выражение. В главе 8 приводится более точное регулярное выражение, которое может использоваться для поиска адресов URL.

Решение

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
string resultString = Regex.Match(subjectString,
                      "http://([a-z0-9.-]+)").Groups[1].Value;
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex("http://([a-z0-9.-]+)");
string resultString = regexObj.Match(subjectString).Groups[1].Value;
```

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim ResultString = Regex.Match(SubjectString,
                   "http://([a-z0-9.-]+)").Groups(1).Value
```

В случае многократного использования того же самого регулярного выражения предпочтительнее создать объект Regex:

```
Dim RegexObj As New Regex("http://([a-z0-9.-]+)")
Dim ResultString = RegexObj.Match(SubjectString).Groups(1).Value
```

Java

```
String resultString = null;
Pattern regex = Pattern.compile("http://([a-z0-9.-]+)");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    resultString = regexMatcher.group(1);
}
```

JavaScript

```
var result;
var match = /http:\/\/([a-z0-9.-]+)/.exec(subject);
if (match) {
    result = match[1];
} else {
    result = "";
}
```

PHP

```
if (preg_match('%http://([a-z0-9.-]+)%', $subject, $groups)) {
    $result = $groups[1];
} else {
    $result = '';
}
```

Perl

```
if ($subject =~ m!http://([a-z0-9.-]+)!) {
    $result = $1;
} else {
    $result = '';
}
```

Python

Для быстрой однократной проверки можно использовать глобальную функцию:

```
matchobj = re.search("http://([a-z0-9.-]+)", subject)
if matchobj:
```

```
    result = matchobj.group(1)
else:
    result = ""
```

В случае многократного использования того же самого регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile("http://([a-z0-9.-]+)")
matchobj = reobj.search(subject)
if match:
    result = matchobj.group(1)
else:
    result = ""
```

Ruby

Можно использовать оператор =~ и его специальные нумерованные переменные, такие как $1:

```
if subject =~ %r!http://([a-z0-9.-]+)!
    result = $1
else
    result = ""
end
```

Как вариант можно задействовать метод `match()` **объекта** `Regexp`**:**

```
matchobj = %r!http://([a-z0-9.-]+)!.match(subject)
if matchobj
    result = matchobj[1]
else
    result = ""
end
```

Обсуждение

В рецептах 2.10 и 2.21 описывалось, как использовать нумерованные обратные ссылки в регулярных выражениях и в замещающем тексте, чтобы можно было описать совпадение с тем же текстом еще раз или вставить часть совпадения с регулярным выражением в замещающий текст. Те же самые номера ссылок можно использовать в программном коде, чтобы извлекать совпадения с сохраняющими группами.

Нумерация сохраняющих групп в регулярных выражениях ведется начиная с единицы. В языках программирования нумерация элементов списков или массивов обычно начинается с нуля. Во всех языках программирования, описываемых в этой книге, где сохраняющие группы запоминаются в массиве или в списке, используется та же нумерация сохраняющих групп, что и в регулярном выражении, то есть начиная с единицы. Нулевой элемент в массиве или в списке используется для сохранения совпадения со всем регулярным выражением. Это означает, что если в регулярном выражении имеется три сохраняющие группы,

массив, хранящий совпадения с ними, будет иметь четыре элемента. Нулевой элемент будет хранить общее совпадение со всем регулярным выражением, а первый, второй и третий элементы – текст, совпавший с тремя сохраняющими группами.

.NET

Чтобы извлечь информацию о сохраняющих группах, потребуется обратиться к одной из версий функции-члена `Regex.Match()`, впервые описанной в рецепте 3.7. Возвращаемый ею объект `Match` имеет свойство `Groups`, которое представляет собой коллекцию типа `GroupCollection`. В коллекции хранится информация обо всех сохраняющих группах, присутствующих в регулярном выражении. Элемент `Groups[1]` хранит информацию о первой сохраняющей группе, элемент `Groups[2]` – о второй группе и т. д.

Коллекция `Groups` хранит по одному объекту `Group` для каждой сохраняющей группы. Класс `Group` имеет те же свойства, что и класс `Match`, за исключением свойства `Groups`. При обращении к `Match.Groups[1].Value` возвращается текст, совпавший с первой сохраняющей группой, точно так же, как при обращении к `Match.Value` возвращается совпадение со всем регулярным выражением. При обращении к `Match.Groups[1].Index` и `Match.Groups[1].Length` возвращаются начальная позиция и длина текста, совпавшего с группой. Подробнее о свойствах `Index` и `Length` говорится в рецепте 3.8.

Элемент `Groups[0]` хранит информацию о совпадении со всем регулярным выражением, которая также хранится в самом объекте совпадения. Свойства `Match.Value` и `Match.Groups[0].Value` полностью эквивалентны.

Коллекция `Groups` не возбуждает исключение при использовании ошибочного номера группы. Например, при обращении к элементу `Groups[-1]` также будет получен объект `Group`, но свойства этого объекта будут свидетельствовать об отсутствии совпадения с этой несуществующей сохраняющей группой под номером `-1`. Лучший способ проверить это – воспользоваться свойством `Success`. При обращении к `Groups[-1].Success` будет возвращено значение `false`.

Чтобы определить количество имеющихся сохраняющих групп, следует проверить свойство `Match.Groups.Count`. Свойство `Count` следует тем же соглашениям, что и свойства `Count` коллекций всех типов в платформе .NET: при обращении к нему возвращается число элементов в коллекции, которое определяется как наибольший индекс плюс один. В данном примере коллекция `Groups` хранит элементы `Groups[0]` и `Groups[1]`. А свойство `Groups.Count` имеет значение `2`.

Java

Программный код, извлекающий текст совпадения с сохраняющей группой или информацию о сохраняющей группе, практически не отличается от программного кода, показанного в предыдущих примерах,

извлекающего ту же информацию о совпадении со всем регулярным выражением. Методы `group()`, `start()` и `end()` класса `Matcher` принимают по одному необязательному аргументу. Без этого аргумента или при передаче в нем нулевого значения они возвращают совпадение или позиции, относящиеся ко всему регулярному выражению.

При передаче положительного числа возвращается информация о соответствующей сохраняющей группе. Нумерация групп начинается с единицы, так же как и нумерация обратных ссылок в самом регулярном выражении. Если указать число, превышающее число сохраняющих групп в регулярном выражении, все три функции возбудят исключение `IndexOutOfBoundsException`. Если сохраняющая группа с указанным номером присутствует в регулярном выражении, но она не участвовала в сопоставлении, метод `group(`*n*`)` вернет значение `null`, а методы `start(`*n*`)` и `end(`*n*`)` вернут -1.

JavaScript

Как описывалось в предыдущем рецепте, метод `exec()` объекта регулярного выражения возвращает массив с информацией о совпадении. Нулевой элемент этого массива хранит совпадение со всем регулярным выражением. Первый элемент хранит текст, совпавший с первой сохраняющей группой, второй элемент – со второй сохраняющей группой и т. д.

Если в строке отсутствует совпадение с регулярным выражением, метод `regexp.exec()` вернет значение `null`.

PHP

В рецепте 3.7 описывалось, как получить текст, совпавший с регулярным выражением, передавая третий аргумент функции `preg_match()`. Когда функция `preg_match()` возвращает значение `1`, в этот аргумент записывается массив. Нулевой элемент этого массива хранит строку совпадения со всем регулярным выражением.

Первый элемент хранит текст, совпавший с первой сохраняющей группой, второй элемент – со второй сохраняющей группой и т. д. Длина массива равна числу сохраняющих групп плюс один. Индексы в массиве соответствуют номерам обратных ссылок в регулярном выражении.

Если в четвертом аргументе функции `preg_match()` передать константу `PREG_OFFSET_CAPTURE`, как описывалось в предыдущем рецепте, длина массива по-прежнему будет равна числу сохраняющих групп плюс один. Но вместо строк в каждом элементе массива будет храниться подмассив из двух элементов. Нулевой подэлемент хранит строку с текстом совпадения для всего регулярного выражения или сохраняющей группы. Первый подэлемент хранит целое число, определяющее позицию начала совпадения в испытуемой строке.

Perl

Когда оператор m// сопоставления с шаблоном находит совпадение, он устанавливает значения нескольких специальных переменных. В число таких переменных входят $1, $2, $3 и т. д., которые хранят части строки, совпавшие с сохраняющими группами в регулярном выражении.

Python

Решение этой задачи во многом идентично решению, что приводится в рецепте 3.7. Только вместо вызова метода group() без параметров следует указать номер интересующей сохраняющей группы. Вызов group(1) вернет текст, совпавший с первой сохраняющей группой, group(2) — со второй группой и т. д. В языке Python поддерживается до 99 сохраняющих групп. Группа с номером 0 представляет совпадение со всем регулярным выражением. Если передать число, превышающее количество сохраняющих групп в регулярном выражении, метод group() возбудит исключение IndexError. Если группа с указанным номером существует, но она не участвовала в сопоставлении, метод group() вернет значение None.

Имеется возможность передавать методу group() сразу несколько номеров сохраняющих групп, чтобы получить совпадения с несколькими группами за одно обращение. В этом случае результатом работы метода будет список строк.

Если необходимо получить кортеж с текстами совпадений для всех сохраняющих групп, можно воспользоваться методом groups() объекта MatchObject. Кортеж будет хранить значения None для тех групп, которые не участвовали в сопоставлении. Если передать методу groups() некоторое значение, оно будет использовано вместо значения None для групп, не участвовавших в сопоставлении.

Если вместо кортежа требуется получить словарь совпадений с сохраняющими группами, вместо метода groups() можно вызвать метод groupdict(). Если передать методу groupdict() некоторое значение, оно будет помещаться в словарь вместо значения None для групп, не участвовавших в сопоставлении.

Ruby

В рецепте 3.8 описывались переменная $~ и объект MatchData. В контексте, где ожидается массив, этот объект интерпретируется как массив совпадений со всеми сохраняющими группами, имеющимися в регулярном выражении. Нумерация сохраняющих групп начинается с 1, как и обратных ссылок в регулярном выражении. Нулевой элемент массива хранит совпадение со всем регулярным выражением.

Переменные $1, $2 и т. д. — это специальные переменные, доступные только для чтения. $1 — это сокращенная форма записи для $~[1]; в этой переменной хранится текст совпадения с первой сохраняющей группой. В переменной $2 хранится текст совпадения со второй группой и т. д.

Именованное сохранение

Если в регулярном выражении используются именованные сохраняющие группы, для получения совпадений в программном коде можно использовать их имена.

C#

Для быстрой однократной проверки можно использовать статическую функцию:

```
string resultString = Regex.Match(subjectString,
                      "http://(?<domain>[a-z0-9.-]+)").Groups["domain"].Value;
```

В случае многократного использования того же регулярного выражения предпочтительнее создать объект Regex**:**

```
Regex regexObj = new Regex("http://(?<domain>[a-z0-9.-]+)");
string resultString = regexObj.Match(subjectString).Groups["domain"].Value;
```

В языке C# нет отличий между способом получения объекта Group **для нумерованной и именованной групп. В последнем случае коллекция** Groups **индексируется не целыми числами, а строками. Кроме того, в данном случае платформа .NET не будет возбуждать исключение, если указанная группа не существует. При обращении к** Match.Groups["nosuchgroup"].Success **просто будет возвращено значение** false**.**

VB.NET

Для быстрой однократной проверки можно использовать статическую функцию:

```
Dim ResultString = Regex.Match(SubjectString,
                   "http://(?<domain>[a-z0-9.-]+)").Groups("domain").Value
```

В случае многократного использования того же регулярного выражения предпочтительнее создать объект Regex**:**

```
Dim RegexObj As New Regex("http://(?<domain>[a-z0-9.-]+)")
Dim ResultString = RegexObj.Match(SubjectString).Groups("domain").Value
```

В языке VB.NET нет отличий между способом получения объекта Group **для нумерованной и именованной групп. В последнем случае коллекция** Groups **индексируется не целыми числами, а строками. Кроме того, в данном случае платформа .NET не будет возбуждать исключение, если указанная группа не существует. При обращении к** Match.Groups["nosuchgroup"].Success **просто будет возвращено значение** false**.**

Java

```
String resultString = null;
Pattern regex = Pattern.compile("http://(?<domain>[a-z0-9.-]+)");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
```

```
        resultString = regexMatcher.group("domain");
    }
```

В Java 7 была добавлена поддержка именованных сохраняющих групп. В этой версии также появился еще один перегруженный метод `Matcher.group()`**, принимающий в виде параметра имя сохраняющей группы и возвращающий текст, совпавший с этой группой. Если передать имя несуществующей группы, он возбудит исключение** `IllegalArgumentException`**.**

К сожалению, методы `Matcher.start()` **и** `Matcher.end()` **не имеют похожих перегруженных версий. Если требуется получить позицию начала и конца совпадения с именованной сохраняющей группой, необходимо указать ее порядковый номер. В языке Java используется сквозная нумерация именованных и неименованных групп, слева направо. Методы** `group()`**,** `start()` **и** `end()` **класса** `Matcher` **принимают единственный необязательный параметр. При вызове без параметра или со значением** `0` **в параметре они вернут совпадение со всем регулярным выражением или соответствующие ему позиции.**

XRegExp

```
var result;
var match = XRegExp.exec(subject,
                         XRegExp("http://(?<domain>[a-z0-9.-]+)"));
if (match) {
    result = match.domain;
} else {
    result = "";
}
```

Библиотека XRegExp добавляет поддержку именованных групп в синтаксис регулярных выражений языка JavaScript. Вызов метода `XRegExp.exec()` **добавляет в возвращаемый объект** `match` **одноименное свойство для каждой именованной группы, что дает возможность ссылаться на группы по их именам.**

PHP

```
if (preg_match('%http://(?P<domain>[a-z0-9.-]+)%', $subject, $groups)) {
    $result = $groups['domain'];
} else {
    $result = '';
}
```

Если в регулярном выражении имеются именованные сохраняющие группы, переменной `$groups` **будет присвоен ассоциативный массив. Текст совпадения для каждой сохраняющей группы будет добавлен в массив дважды. Извлекать совпадения из массива можно с использованием как номеров групп, так и их имен. В примере программного кода элемент** `$groups[0]` **хранит совпадение со всем регулярным выражением, тогда как элементы** `$groups[1]` **и** `$groups['domain']` **хранят текст совпадения с единственной в этом регулярном выражении сохраняющей группой.**

Perl

```
if ($subject =~ m!http://(?<domain>[a-z0-9.-]+)!) {
    $result = $+{'domain'};
} else {
    $result = '';
}
```

Именованная сохраняющая группировка поддерживается в языке Perl начиная с версии 5.10. В хеше %+ хранятся тексты совпадений со всеми именованными сохраняющими группами. В языке Perl именованные группы нумеруются вместе с нумерованными группами. В данном примере переменные $1 и $+{name} хранят текст совпадения с единственной в этом регулярном выражении сохраняющей группой.

Python

```
matchobj = re.search("http://(?P<domain>[a-z0-9.-]+)", subject)
if matchobj:
    result = matchobj.group("domain")
else:
    result = ""
```

Если в регулярном выражении имеются именованные сохраняющие группы, то вместо номера методу group() можно передавать имена групп.

Ruby

В Ruby 1.9 добавлена поддержка именованных сохраняющих групп. Поддержка именованных сохранений в этой версии также была добавлена в переменную $~ и объект MatchData, описанный в рецепте 3.8. Обращение к $~["name"] или matchobj["name"] возвращает текст, совпавший с именованной группой «name». Позицию начала и конца совпадения с именованной группой можно получить вызовами matchobj.begin("name") и matchobj.end("name").

См. также

Рецепт 2.9, где описываются нумерованные сохраняющие группы.

Рецепт 2.11, где описываются именованные сохраняющие группы.

3.10. Извлечение списка всех совпадений

Задача

Во всех предыдущих рецептах этой главы речь шла только о первом совпадении, обнаруженном регулярным выражением в испытуемой строке. Но во многих случаях регулярное выражение, совпадающее с частью строки, может обнаруживать еще одно совпадение в оставшейся части строки. А за вторым совпадением может следовать третье и т. д.

Например, в строке `The lucky numbers are 7, 13, 16, 42, 65, and 99` **регулярное выражение** `‹\d+›` **может обнаружить шесть совпадений: 7, 13, 16, 42, 65 и 99.**

Необходимо извлечь список всех подстрок, которые обнаруживаются регулярным выражением при многократном применении его к фрагменту строки, остающемуся после каждого очередного совпадения.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
MatchCollection matchlist = Regex.Matches(subjectString, @"\d+");
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`**:**

```
Regex regexObj = new Regex(@"\d+");
MatchCollection matchlist = regexObj.Matches(subjectString);
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim MatchList = Regex.Matches(SubjectString, "\d+")
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`**:**

```
Dim RegexObj As New Regex("\d+")
Dim MatchList = RegexObj.Matches(SubjectString)
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    resultList.add(regexMatcher.group());
}
```

JavaScript

```
var list = subject.match(/\d+/g);
```

PHP

```
preg_match_all('/\d+/', $subject, $result, PREG_PATTERN_ORDER);
$result = $result[0];
```

Perl

```
@result = $subject =~ m/\d+/g;
```

Этот прием работает, только если регулярное выражение не содержит сохраняющих групп, поэтому в подобных случаях следует использовать несохраняющие группы. Подробности приводятся в рецепте 2.9.

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
result = re.findall(r"\d+", subject)
```

В случае многократного использования того же регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"\d+")
result = reobj.findall(subject)
```

Ruby

```
result = subject.scan(/\d+/)
```

Обсуждение

.NET

Метод `Matches()` класса `Regex` многократно применяет регулярное выражение к строке, пока не будут найдены все совпадения. Он возвращает объект `MatchCollection`, в котором хранятся все совпадения. Первым аргументом всегда передается испытуемая строка. В этой строке будут разыскиваться совпадения с регулярным выражением. Первый аргумент не может иметь значение `null`. В противном случае метод `Matches()` возбудит исключение `ArgumentNullException`.

Если требуется выполнить поиск совпадений с регулярным выражением в небольшом количестве строк, можно использовать статическую перегруженную версию метода `Matches()`. В первом аргументе этому методу передается испытуемая строка, а во втором – регулярное выражение. Параметры регулярного выражения можно передавать в третьем необязательном аргументе.

Если предстоит обработать большое число строк, предпочтительнее сначала создать объект `Regex` и затем использовать его для вызова метода `Matches()`. Единственным обязательным аргументом для этого метода является испытуемая строка. Во втором необязательном аргументе можно передать номер позиции символа, с которого должны начинаться попытки сопоставления с регулярным выражением. Фактически число, указываемое во втором аргументе, – это количество символов от начала испытуемой строки, которые должны игнорироваться регуляр-

ным выражением. Это может быть удобно, когда часть строки уже была проверена и необходимо выполнить проверку в оставшейся части строки. Указываемое число должно быть между нулем и длиной строки. В противном случае метод `Matches()` возбудит исключение `ArgumentOutOfRangeException`.

Статические перегруженные версии методов не имеют аргумента, с помощью которого можно было бы указать, с какой позиции в строке следует начинать поиск совпадения. Не существует перегруженной версии метода `Matches()`, которая позволяла бы указывать, в какой позиции перед концом строки следует останавливать поиск. В случае необходимости можно вызывать метод `Regex.Match("subject", start, stop)` в цикле, как показано в следующем рецепте, и вручную добавлять в список все найденные совпадения.

Java

В языке Java отсутствует функция, которая извлекала бы список совпадений. Но ее легко можно реализовать, адаптировав решение из рецепта 3.7. Для этого вызов метода `find()` нужно производить не в условном операторе `if`, а в цикле `while`.

Чтобы использовать классы `List` и `ArrayList`, как в этом примере, нужно добавить инструкцию `import java.util.*;` в начало файла с программным кодом.

JavaScript

В данном решении вызывается метод `string.match()` точно так же, как в решении к рецепту 3.7. Но здесь есть одно маленькое, но важное отличие: флаг `/g`. Подробнее о флагах регулярных выражений рассказывается в рецепте 3.4.

Флаг `/g` предписывает функции `match()` выполнить итерации по всем совпадениям в строке и поместить их в массив. В приведенном фрагменте программного кода элемент `list[0]` будет хранить первое совпадение с регулярным выражением, `list[1]` – второе и т. д. Узнать количество совпадений можно с помощью свойства `list.length`. В случае отсутствия совпадений метод `string.match()` как обычно вернет значение `null`.

Элементами этого массива являются строки. При использовании флага `/g` метод `string.match()` не предоставляет никакой дополнительной информации о совпадениях с регулярным выражением. Если необходимо получить дополнительные сведения о совпадениях, следует выполнить итерации по совпадениям, как описывается в рецепте 3.11.

PHP

Во всех решениях на языке PHP в предыдущих рецептах использовалась функция `preg_match()`, которая отыскивает первое совпадение с регулярным выражением в строке. Функция `preg_match_all()` очень похо-

жа на нее. Главное ее отличие состоит в том, что она отыскивает в строке все совпадения и возвращает целое число, обозначающее число найденных совпадений с регулярным выражением.

Первые три аргумента функции `preg_match_all()` идентичны первым трем аргументам функции `preg_match()`: строка с регулярным выражением, строка, в которой следует произвести поиск, и переменная, в которую будет записан массив с результатами. Единственное отличие состоит в том, что третий аргумент является обязательным, а массив всегда многомерный.

В четвертом аргументе допускается передавать либо значение константы `PREG_PATTERN_ORDER`, либо `PREG_SET_ORDER`. Если опустить четвертый аргумент, по умолчанию будет использоваться значение `PREG_PATTERN_ORDER`.

При использовании значения `PREG_PATTERN_ORDER` будет возвращаться массив, хранящий информацию об общем совпадении в нулевом элементе, а информацию о сохраняющих группах – в первом и последующих элементах. Массив будет иметь длину, равную числу сохраняющих групп плюс один. Тот же порядок построения массива используется в функции `preg_match()`. Разница заключается в том, что вместо каждого элемента, хранящего строку с единственным совпадением, найденным функцией `preg_match()`, используются подмассивы со всеми совпадениями, найденными функцией `preg_match_all()`. Длина каждого подмассива совпадает со значением, возвращаемым функцией `preg_match_all()`.

Чтобы получить список всех совпадений с регулярным выражением в строке, отбросив совпадения с сохраняющими группами, следует указать значение `PREG_PATTERN_ORDER` и извлечь нулевой элемент массива. Если интерес представляет только текст совпадения с определенной сохраняющей группой, можно использовать значение `PREG_PATTERN_ORDER` и номер сохраняющей группы. Например, после вызова `preg_match_all('%http://([a-z0-9.-]+)%', $subject, $result)` элемент `$result[1]` будет хранить список доменных имен всех адресов URL, имеющихся в испытуемой строке.

Если указано значение `PREG_SET_ORDER`, массив заполняется теми же строками, но несколько иным способом. Длина массива равна значению, возвращаемому функцией `preg_match_all()`. Каждый элемент массива представляет собой подмассив, нулевой элемент которого содержит совпадение со всем регулярным выражением, а первый и последующие элементы – совпадения с сохраняющими группами. Когда указано значение `PREG_SET_ORDER`, элемент `$result[0]` будет хранить тот же самый массив, что возвращает функция `preg_match()`.

Допускается комбинировать флаг `PREG_OFFSET_CAPTURE` с флагом `PREG_PATTERN_ORDER` или `PREG_SET_ORDER`. В этом случае возникает тот же эффект, что и при передаче флага `PREG_OFFSET_CAPTURE` функции `preg_match()` в четвертом аргументе. То есть в каждом элементе массива содержится не строка, а массив из двух элементов, в которых хранятся строка совпадения и смещение этого совпадения от начала испытуемой строки.

Perl

В рецепте 3.4 говорилось, что необходимо добавить модификатор /g, чтобы обеспечить возможность поиска в испытуемой строке нескольких совпадений с регулярным выражением. При использовании регулярного выражения с модификатором /g в списочном контексте оно будет отыскивать и возвращать все совпадения в испытуемой строке. В данном рецепте списочный контекст обеспечивается переменной-списком, расположенной слева от оператора присваивания.

Если в регулярном выражении отсутствуют какие-либо сохраняющие группы, список будет содержать только совпадения со всем регулярным выражением. Если в регулярном выражении присутствуют сохраняющие группы, список будет содержать совпадения со всеми сохраняющими группами для каждого совпадения с регулярным выражением. Совпадение со всем регулярным выражением не включается в список, если выражение целиком не заключить в сохраняющую группу. Если необходимо получить список всех совпадений с регулярным выражением целиком, все сохраняющие группы следует заменить несохраняющими. Обе разновидности группировки описываются в рецепте 2.9.

Python

Поиск всех совпадений с регулярным выражением выполняет функция `findall()` из модуля `re`. Первым аргументом функции передается регулярное выражение, а вторым – испытуемая строка. Параметры регулярного выражения можно передать функции в третьем необязательном аргументе.

Функция `re.findall()` вызывает функцию `re.compile()`, после чего вызывает метод `findall()` объекта скомпилированного регулярного выражения. Этот метод принимает единственный обязательный аргумент: испытуемую строку.

Метод `findall()` имеет два необязательных аргумента, которые не поддерживаются глобальной функцией `re.findall()`. Вслед за испытуемой строкой можно передать позицию символа в строке, с которого метод `findall()` должен начинать поиск. Если опустить этот аргумент, поиск будет выполняться методом `findall()` во всей строке. Если указана начальная позиция поиска, можно также определить конечную позицию. Если конечная позиция не определена, поиск будет выполняться до конца строки.

Независимо от того, с какими аргументами будет вызываться метод `findall()`, в результате всегда будет возвращаться список всех найденных совпадений. Если в регулярном выражении имеются сохраняющие группы, метод вернет список кортежей, содержащих совпадения для всех сохраняющих групп для каждого совпадения с регулярным выражением.

Ruby

Метод `scan()` класса `String` принимает регулярное выражение в виде единственного аргумента. Он отыскивает в строке все совпадения с регулярным выражением. Когда метод `scan()` вызывается без блока, он возвращает массив всех совпадений с регулярным выражением.

Если в регулярном выражении отсутствуют сохраняющие группы, метод `scan()` возвратит массив строк. Каждый элемент этого массива соответствует совпадению с регулярным выражением и содержит текст совпадения.

При наличии сохраняющих групп метод `scan()` вернет массив массивов. Каждый элемент этого массива соответствует совпадению с регулярным выражением и представляет собой массив с совпадениями для всех сохраняющих групп. В нулевом подэлементе будет храниться совпадение с первой сохраняющей группой, в первом – со второй сохраняющей группой и т. д. Совпадения со всем регулярным выражением не включаются в массив. Если необходимо включить совпадения со всем регулярным выражением, следует все регулярное выражение заключить в одну сохраняющую группу.

В языке Ruby отсутствует возможность заставить метод `scan()` возвращать массив строк, когда регулярное выражение содержит сохраняющие группы. Единственное решение заключается в том, чтобы заменить все именованные и нумерованные сохраняющие группы на несохраняющие.

См. также

Рецепт 3.7, где показано, как получить фрагмент, фактически совпавший с регулярным выражением.

Рецепт 3.11, где демонстрируется код, выполняющий обход всех совпадений, найденных регулярным выражением в испытуемой строке.

Рецепт 3.12, где демонстрируется код, выполняющий обход всех совпадений, найденных регулярным выражением в испытуемой строке, и сохраняющий только те, которые соответствуют определенным критериям.

3.11. Обход всех совпадений в цикле

Задача

В предыдущем рецепте рассказывалось, как многократно применить регулярное выражение к строке, чтобы получить список совпадений. Теперь требуется выполнить обход всех совпадений в цикле.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Match matchResult = Regex.Match(subjectString, @"\d+");
while (matchResult.Success) {
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в переменной matchResult
    matchResult = matchResult.NextMatch();
}
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex**:**

```
Regex regexObj = new Regex(@"\d+");
matchResult = regexObj.Match(subjectString);
while (matchResult.Success) {
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в переменной matchResult
    matchResult = matchResult.NextMatch();
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim MatchResult = Regex.Match(SubjectString, "\d+")
While MatchResult.Success
     'Здесь можно выполнить обработку очередного совпадения,
     'хранящегося в переменной matchResult
    MatchResult = MatchResult.NextMatch
End While
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex**:**

```
Dim RegexObj As New Regex("\d+")
Dim MatchResult = RegexObj.Match(SubjectString)
While MatchResult.Success
     'Здесь можно выполнить обработку очередного совпадения,
     'хранящегося в переменной matchResult
    MatchResult = MatchResult.NextMatch
End While
```

Java

```
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    // Здесь можно выполнить обработку очередного совпадения,
```

```
    // хранящегося в объекте regexMatcher
}
```

JavaScript

Если совпадения с регулярным выражением могут иметь нулевую длину или когда это заранее неизвестно, следует предусмотреть обработку проблем, связанных с различной для разных браузеров реакцией метода `exec()` **на совпадения с нулевой длиной:**

```
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Не дать возможность браузерам попасть
    // в бесконечный цикл
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в переменной match
}
```

Если заранее известно, что для регулярного выражения никогда не будут обнаруживаться совпадения нулевой длины, можно выполнять итерации непосредственно по регулярному выражению:

```
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в переменной match
}
```

XRegExp

При использовании библиотеки XRegExp для обхода всех совпадений можно применить специализированный метод `XRegExp.forEach()`**:**

```
XRegExp.forEach(subject, /\d+/, function(match) {
    // Здесь можно выполнить обработку очередного совпадения,
    // хранящегося в аргументе match
});
```

PHP

```
preg_match_all('/\d+/', $subject, $result, PREG_PATTERN_ORDER);
for ($i = 0; $i < count($result[0]); $i++) {
    # Текст совпадения = $result[0][$i];
}
```

Perl

```
while ($subject =~ m/\d+/g) {
    # текст совпадения = $&
}
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
for matchobj in re.finditer(r"\d+", subject):
    # Здесь можно выполнить обработку очередного совпадения,
    # хранящегося в переменной matchobj
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"\d+")
for matchobj in reobj.finditer(subject):
    # Здесь можно выполнить обработку очередного совпадения,
    # хранящегося в переменной matchobj
```

Ruby

```
subject.scan(/\d+/) {|match|
    # Здесь можно выполнить обработку очередного совпадения,
    # хранящегося в переменной match
}
```

Обсуждение

.NET

В рецепте 3.7 описывалось, как с помощью функции-члена `Match()` класса `Regex` получить первое совпадение с регулярным выражением. Чтобы обойти в цикле все совпадения, имеющиеся в строке, необходимо снова вызвать функцию `Match()` для получения информации о первом совпадении. Функция `Match()` возвращает экземпляр класса `Match`, который в представленном решении сохраняется в переменной `matchResult`. Если свойство `Success` объекта `matchResult` содержит значение `true`, можно начинать итерации.

В начале цикла можно использовать свойства объекта `Match` для получения информации о первом совпадении. В рецепте 3.7 описывается свойство `Value`, в рецепте 3.8 – свойства `Index` и `Length` и в рецепте 3.9 – свойство-коллекция `Groups`.

Закончив работу с первым совпадением, можно вызвать функцию-член `NextMatch()` относительно переменной `matchResult`. Функция `Match.NextMatch()` возвращает экземпляр класса `Match`, как и функция `Regex.Match()`. Вновь созданный экземпляр хранит информацию о втором совпадении.

Присваивание результата, полученного при вызове `matchResult.NextMatch()`, одной и той же переменной `matchResult` упрощает обход всех совпадений. Каждый раз в цикле выполняется проверка значений свойства `matchResult.Success`, чтобы убедиться, что функция `NextMatch()` действительно обнаружила еще одно совпадение. Когда функция `NextMatch()`

терпит неудачу, она все равно возвращает объект Match, но в этом случае его свойство Success будет иметь значение false. Используя одну и ту же переменную matchResult, нам удалось совместить в одной инструкции while проверку наличия первого совпадения с проверками наличия всех последующих совпадений после обращений к функции NextMatch().

Функция NextMatch() не изменяет объект Match, относительно которого она вызывается. В случае необходимости можно сохранять объекты Match для каждого совпадения с регулярным выражением.

Метод NextMatch() не имеет аргументов. Он использует то же регулярное выражение и испытуемую строку, которые передавались методу Regex.Match(). Объект Match сохраняет ссылки на регулярное выражение и испытуемую строку.

Статическую версию метода Regex.Match() можно использовать, даже когда испытуемая строка содержит очень большое число совпадений с регулярным выражением. Метод Regex.Match() скомпилирует регулярное выражение всего один раз, а возвращаемый объект Match будет хранить ссылку на скомпилированное регулярное выражение. Метод Match.NextMatch() использует это скомпилированное ранее регулярное выражение, на которое ссылается объект Match, даже если вызывался статический метод Regex.Match(). Создавать экземпляр класса Regex действительно необходимо, только если метод Regex.Match() требуется вызывать многократно (например, чтобы применить одно и то же регулярное выражение к большому числу строк).

Java

Обход всех совпадений в языке Java реализуется очень просто. Достаточно просто в цикле while вызывать метод find(), представленный в рецепте 3.7. Каждый вызов метода find() обновляет информацию, хранящуюся в объекте Matcher, сведениями о совпадении и о позиции, откуда должна начинаться следующая попытка сопоставления.

JavaScript

Если возникнет потребность использовать регулярное выражение в цикле, не забудьте указать флаг /g. Назначение этого флага разъяснялось в рецепте 3.4. Цикл while (regexp.exec()) отыскивает все числа в испытуемой строке, когда regexp = /\d+/g. Если бы регулярное выражение имело вид /\d+/, то цикл while (regexp.exec()) находил бы первое число в строке снова и снова, пока не произошло бы аварийное завершение сценария или пока сценарий не был бы принудительно завершен браузером.

Примечательно, что цикл while (/\d+/g.exec()) (в котором используется литерал регулярного выражения с флагом /g) также превратился бы в бесконечный цикл, по крайней мере, в некоторых реализациях JavaScript, потому что регулярное выражение компилировалось бы заново в каждой итерации цикла while. Когда выполняется компиляция регулярного выражения, начальная позиция для предстоящей попытки со-

поставления сбрасывается в начало строки. Присваивание регулярного выражения переменной за пределами цикла гарантирует, что оно будет скомпилировано всего один раз.

Объект, возвращаемый методом `regexp.exec()`, описывается в рецептах 3.8 и 3.9. Это один и тот же объект, даже когда метод `exec()` вызывается в цикле. Допускается делать с этим объектом все, что потребуется.

Единственный эффект действия флага `/g` заключается в обновлении значения свойства `lastIndex` объекта `regexp`, чей метод `exec()` вызывается. Этот эффект действует даже при использовании литерала регулярного выражения, как показано во втором решении для этого рецепта. При каждом последующем обращении к методу `exec()` попытка сопоставления начинается с позиции `lastIndex`. Если присвоить свойству `lastIndex` новое значение, попытка сопоставления начнется с указанной позиции.

Однако со свойством `lastIndex` связана одна очень существенная проблема. Если понимать стандарт ECMA-262v3 для JavaScript буквально, то метод `exec()` должен записывать в свойство `lastIndex` позицию первого символа, следующего за совпадением. Это означает, что когда совпадение имеет нулевую длину, следующая попытка сопоставления должна начинаться с позиции, где только что было найдено совпадение, что в результате приводит к бесконечному циклу.

Все современные браузеры реализуют стандарт так, как написано, а это означает, что при использовании метода `regexp.exec()` можно попасть в бесконечный цикл. А ведь такой исход весьма вероятен. Например, для обхода всех строк в многострочном тексте можно использовать инструкции `re = /^.*$/gm; while (re.exec())`, но если в тексте имеется пустая строка, сценарий «споткнется» о нее.

Решение проблемы состоит в том, чтобы увеличивать на 1 значение свойства `lastIndex`, если функция `exec()` этого еще не сделала. Как это делается, показано в первом решении для JavaScript. Если вы не знаете, как могут развиваться события, просто вставьте эту строку программного кода и вы будете застрахованы от этой проблемы.

В старых версиях Internet Explorer эта проблема решалась увеличением свойства `lastIndex` на единицу, если совпадение имеет нулевую длину. В Internet Explorer 9 это необходимо делать только при работе в режиме совместимости (quirks mode). Именно по этой причине в рецепте 3.7 говорилось о невозможности использовать свойство `lastIndex` для определения конца совпадения – Internet Explorer возвращает неправильные значения при работе в режиме совместимости.

Все остальные механизмы регулярных выражений, рассматриваемые в этой книге, решают эту проблему, автоматически начиная следующую попытку сопоставления со следующего символа в строке, если предыдущее совпадение имело нулевую длину.

Данная проблема не проявляется при использовании метода `string.replace()` (рецепт 3.14) или при поиске всех совпадений с помощью метода

string.match() (рецепт 3.10). Для этих методов, внутренняя реализация которых использует свойство lastIndex, стандарт ECMA-262v3 оговаривает, что значение lastIndex должно увеличиваться на 1 при каждом совпадении нулевой длины.

XRegExp

При использовании JavaScript-библиотеки XRegExp можно здорово облегчить себе жизнь, применяя специализированный метод XRegExp.forEach(). Передайте этому методу испытуемую строку, регулярное выражение и функцию обратного вызова, и он вызовет вашу функцию для каждого найденного совпадения. Функции обратного вызова будут переданы в виде параметров: массив совпадений, индекс текущего совпадения (счет начинается с нуля), испытуемая строка и регулярное выражение. Если методу XRegExp.forEach() передать четвертый параметр, он будет использоваться как контекст для передачи функции обратного вызова и будет возвращен методом XRegExp.forEach() после завершения поиска совпадений.

Метод XRegExp.forEach() игнорирует свойства global и lastIndex передаваемого ему объекта RegExp. Он всегда выполняет итерации по всем совпадениям. Используйте XRegExp.forEach(), чтобы надежно решить любые проблемы, связанные с совпадениями нулевой длины.

В библиотеке XRegExp также имеется собственный метод XRegExp.exec(). Он игнорирует свойство lastIndex. Вместо этого в третьем необязательном параметре ему можно передать позицию, с которой следует начинать попытки сопоставления. Чтобы найти следующее совпадение, достаточно указать позицию, где закончилось предыдущее совпадение. Если оно имеет нулевую длину, следует передать позицию конца предыдущего совпадения плюс 1.

PHP

Функция preg_match() принимает пятый необязательный аргумент, определяющий позицию в строке, откуда следует начинать попытки сопоставления. Для решения поставленной задачи можно было бы адаптировать решение из рецепта 3.8 и передавать функции preg_match() сумму $matchstart + $matchlength в виде пятого аргумента при попытке найти второе совпадение в строке и продолжать повторять то же самое в третьей и последующих попытках, пока preg_match() не вернет 0. Этот прием используется в рецепте 3.18.

В дополнение к необходимости вычислять в программном коде начальное смещение для каждой попытки сопоставления многократное обращение к функции preg_match() само по себе неэффективно из-за отсутствия возможности сохранять скомпилированное регулярное выражение в переменной. Функция preg_match() вынуждена искать скомпилированное регулярное выражение в своем кэше при каждом вызове.

Более простое и более эффективное решение заключается в использовании функции `preg_match_all()`, как описано в предыдущем рецепте, и выполнении итераций в массиве с результатами сопоставления.

Perl

В рецепте 3.4 говорилось, что в регулярное выражение необходимо добавить модификатор `/g`, чтобы отыскать в испытуемой строке все совпадения. Если регулярное выражение с модификатором `/g` используется в скалярном контексте, следующая попытка сопоставления будет начинаться в конце предыдущего совпадения. В этом рецепте инструкция `while` обеспечивает скалярный контекст. Все специальные переменные, такие как `$&` (описываются в рецепте 3.7), существуют только в области видимости цикла `while`.

Python

Функция `finditer()` из модуля `re` возвращает итератор, который может использоваться для поиска всех совпадений с регулярным выражением. Первым аргументом эта функция принимает регулярное выражение, а вторым – испытуемую строку. Параметры регулярного выражения могут передаваться в третьем необязательном аргументе.

Функция `re.finditer()` вызывает функцию `re.compile()`, после чего вызывает метод `finditer()` объекта скомпилированного регулярного выражения. Этот метод имеет единственный обязательный аргумент – испытуемую строку.

Метод `finditer()` принимает два необязательных аргумента, которые не поддерживаются глобальной функцией `re.finditer()`. Вслед за аргументом с испытуемой строкой можно передать позицию символа в строке, откуда метод `finditer()` должен начинать поиск. Если этот аргумент отсутствует, поиск будет выполняться во всей строке. Если указана начальная позиция, то можно указать и конечную позицию. Если конечная позиция не определена, поиск выполняется до конца строки.

Ruby

Метод `scan()` класса `String` принимает регулярное выражение в виде единственного аргумента и выполняет обход всех совпадений с регулярным выражением в строке. При вызове с блоком появляется возможность обрабатывать каждое совпадение, как только оно будет найдено.

Если регулярное выражение не содержит сохраняющих групп, в блоке следует указать только одну переменную цикла. Эта переменная будет принимать текст совпадения с регулярным выражением.

Если регулярное выражение содержит одну или более сохраняющих групп, необходимо указать по одной переменной для каждой группы. Первая переменная будет принимать текст совпадения с первой сохраняющей группой, вторая переменная – со второй сохраняющей группой

и т. д. Ни в одну из переменных не будет записываться совпадение со всем регулярным выражением. Если необходимо включить совпадение со всем регулярным выражением, его следует заключить в дополнительную сохраняющую группу.

```
subject.scan(/(a)(b)(c)/) {|a, b, c|
    # a, b и c хранят совпадения с тремя сохраняющими группами
}
```

Если переменных указано меньше, чем сохраняющих групп в регулярном выражении, получить доступ можно будет только к группам, для которых имеются переменные. Если переменных указано больше, чем сохраняющих групп, в лишние переменные будет записываться значение `nil`.

Если указана только одна переменная цикла, тогда как в регулярном выражении имеется одна или более сохраняющих групп, переменная будет заполняться массивом строк. Каждая строка в массиве будет соответствовать сохраняющей группе. Если в выражении имеется всего одна сохраняющая группа, массив будет состоять из единственного элемента:

```
subject.scan(/(a)(b)(c)/) {|abc|
    # в элементах abc[0], abc[1] и abc[2] хранится текст
    # совпадений с тремя сохраняющими группами
}
```

См. также

Рецепт 3.12, который дополняет этот рецепт и демонстрирует извлечение только совпадений, соответствующих определенным критериям.

Рецепт 3.7, где показано, как получить только первое совпадение с регулярным выражением.

Рецепт 3.8, где демонстрируется код, определяющий позицию и длину совпадения.

Рецепт 3.10, где демонстрируется код, извлекающий список всех совпадений, которые были найдены регулярным выражением в испытуемой строке.

Рецепт 3.22, где рассказывается, как сконструировать простой парсер, выполняющий итерации по всем совпадениям с регулярным выражением.

3.12. Проверка полученных совпадений в программном коде

Задача

В рецепте 3.10 показано, как получить список всех совпадений с регулярным выражением, применяя его к фрагменту строки, остающемуся

после каждого сопоставления. Теперь требуется получить список совпадений, удовлетворяющих некоторому критерию, который невозможно (достаточно просто) выразить в виде регулярного выражения. Например, при получении списка счастливых номеров требуется оставить только те, которые без остатка делятся на 13.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
StringCollection resultList = new StringCollection();
Match matchResult = Regex.Match(subjectString, @"\d+");
while (matchResult.Success) {
    if (int.Parse(matchResult.Value) % 13 == 0) {
        resultList.Add(matchResult.Value);
    }
    matchResult = matchResult.NextMatch();
}
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
StringCollection resultList = new StringCollection();
Regex regexObj = new Regex(@"\d+");
matchResult = regexObj.Match(subjectString);
while (matchResult.Success) {
    if (int.Parse(matchResult.Value) % 13 == 0) {
        resultList.Add(matchResult.Value);
    }
    matchResult = matchResult.NextMatch();
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim ResultList = New StringCollection
Dim MatchResult = Regex.Match(SubjectString, "\d+")
While MatchResult.Success
    If Integer.Parse(MatchResult.Value) Mod 13 = 0 Then
            ResultList.Add(MatchResult.Value)
    End If
    MatchResult = MatchResult.NextMatch
End While
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Dim ResultList = New StringCollection
Dim RegexObj As New Regex("\d+")
```

```
Dim MatchResult = RegexObj.Match(SubjectString)
While MatchResult.Success
    If Integer.Parse(MatchResult.Value) Mod 13 = 0 Then
        ResultList.Add(MatchResult.Value)
    End If
    MatchResult = MatchResult.NextMatch
End While
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    if (Integer.parseInt(regexMatcher.group()) % 13 == 0) {
        resultList.add(regexMatcher.group());
    }
}
```

JavaScript

```
var list = [];
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Не позволять браузерам, попадать в бесконечный цикл
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Здесь можно обработать совпадение, хранящееся в переменной match
    if (match[0] % 13 == 0) {
        list.push(match[0]);
    }
}
```

XRegExp

```
var list = [];
XRegExp.forEach(subject, /\d+/, function(match) {
    if (match[0] % 13 == 0) {
        list.push(match[0]);
    }
});
```

PHP

```
preg_match_all('/\d+/', $subject, $matchdata, PREG_PATTERN_ORDER);
for ($i = 0; $i < count($matchdata[0]); $i++) {
    if ($matchdata[0][$i] % 13 == 0) {
        $list[] = $matchdata[0][$i];
    }
}
```

Perl

```
while ($subject =~ m/\d+/g) {
    if ($& % 13 == 0) {
```

```
        push(@list, $&);
    }
}
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
list = []
for matchobj in re.finditer(r"\d+", subject):
    if int(matchobj.group()) % 13 == 0:
        list.append(matchobj.group())
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект скомпилированного регулярного выражения:

```
list = []
reobj = re.compile(r"\d+")
for matchobj in reobj.finditer(subject):
    if int(matchobj.group()) % 13 == 0:
        list.append(matchobj.group())
```

Ruby

```
list = []
subject.scan(/\d+/) {|match|
    list << match if (Integer(match) % 13 == 0)
}
```

Обсуждение

Регулярное выражение имеет дело с текстом. Хотя регулярному выражению ‹\d+› соответствует то, что мы называем числами, для механизма регулярных выражений это просто строки, состоящие из одной или более цифр.

Если требуется отыскать какие-то определенные числа, например числа, кратные 13, намного проще написать универсальное регулярное выражение, отыскивающее совпадения со всеми числами, а затем добавить немного программного кода, который будет отбрасывать совпадения, не представляющие интереса.

Все решения для этого рецепта основаны на решениях из предыдущего рецепта, где демонстрируется, как выполнять обход всех совпадений. Совпадение с регулярным выражением внутри цикла преобразуется в число.

Некоторые языки программирования делают это автоматически, другие требуют явно вызвать функцию, которая преобразует строку в число. Далее проверяется, делится ли целое число на 13 без остатка. Если делится, совпадение добавляется в список. Если нет, совпадение пропускается.

См. также

Рецепт 3.12, использовавшийся в качестве основы для данного рецепта. В нем описывается, как выполнять итерации по найденным совпадениям.

Рецепт 3.7, где показано, как получить только первое совпадение с регулярным выражением.

Рецепт 3.8, где демонстрируется код, определяющий позицию и длину совпадения.

Рецепт 3.10, где демонстрируется код, извлекающий список всех совпадений, которые были найдены регулярным выражением в испытуемой строке.

3.13. Поиск совпадения внутри другого совпадения

Задача

Необходимо отыскать все совпадения с определенным регулярным выражением, но только внутри некоторого раздела испытуемой строки. Имеется еще одно регулярное выражение, которому соответствуют все разделы в строке.

Допустим, что имеется файл HTML, в котором различные сообщения выводятся жирным шрифтом с помощью тегов <b>. Необходимо отыскать все числа, заключенные в этот тег. Если текст, заключенный в тег <b> содержит несколько чисел, все они должны рассматриваться как отдельные совпадения. Например, для строки `1 <b>2</b> 3 4 <b>5 6 7</b>` должно быть найдено четыре совпадения: 2, 5, 6 и 7.

Решение

C#

```
StringCollection resultList = new StringCollection();
Regex outerRegex = new Regex("<b>(.*?)</b>", RegexOptions.Singleline);
Regex innerRegex = new Regex(@"\d+");
// Найти первый раздел
Match outerMatch = outerRegex.Match(subjectString);
while (outerMatch.Success) {
    // Получить совпадения внутри раздела
    Match innerMatch = innerRegex.Match(outerMatch.Groups[1].Value);
    while (innerMatch.Success) {
        resultList.Add(innerMatch.Value);
        innerMatch = innerMatch.NextMatch();
    }
    // Найти следующий раздел
    outerMatch = outerMatch.NextMatch();
}
```

VB.NET

```
Dim ResultList = New StringCollection
Dim OuterRegex As New Regex("<b>(.*?)</b>", RegexOptions.Singleline)
Dim InnerRegex As New Regex("\d+")
'Найти первый раздел
Dim OuterMatch = OuterRegex.Match(SubjectString)
While OuterMatch.Success
    'Получить совпадения внутри раздела
    Dim InnerMatch = InnerRegex.Match(OuterMatch.Groups(1).Value)
    While InnerMatch.Success
        ResultList.Add(InnerMatch.Value)
        InnerMatch = InnerMatch.NextMatch
    End While
    OuterMatch = OuterMatch.NextMatch
End While
```

Java

Проще всего реализовать итерации можно с использованием двух объектов совпадений, и этот прием будет работать в версии Java 4 и выше:

```
List<String> resultList = new ArrayList<String>();
Pattern outerRegex = Pattern.compile("<b>(.*?)</b>", Pattern.DOTALL);
Pattern innerRegex = Pattern.compile("\\d+");
Matcher outerMatcher = outerRegex.matcher(subjectString);
while (outerMatcher.find()) {
    Matcher innerMatcher = innerRegex.matcher(outerMatcher.group(1));
    while (innerMatcher.find()) {
        resultList.add(innerMatcher.group());
    }
}
```

Следующий фрагмент обладает большей эффективностью (потому что объект `innerMatcher` **создается всего один раз), но он будет работать только в версии Java 5 или выше:**

```
List<String> resultList = new ArrayList<String>();
Pattern outerRegex = Pattern.compile("<b>(.*?)</b>", Pattern.DOTALL);
Pattern innerRegex = Pattern.compile("\\d+");
Matcher outerMatcher = outerRegex.matcher(subjectString);
Matcher innerMatcher = innerRegex.matcher(subjectString);
while (outerMatcher.find()) {
    innerMatcher.region(outerMatcher.start(1), outerMatcher.end(1));
    while (innerMatcher.find()) {
        resultList.add(innerMatcher.group());
    }
}
```

JavaScript

```
var result = [];
var outerRegex = /<b>([\s\S]*?)<\/b>/g;
```

```
var innerRegex = /\d+/g;
var outerMatch;
var innerMatches;
while (outerMatch = outerRegex.exec(subject)) {
    if (outerMatch.index == outerRegex.lastIndex)
        outerRegex.lastIndex++;
    innerMatches = outerMatch[1].match(innerRegex);
    if (innerMatches) {
        result = result.concat(innerMatches);
    }
}
```

XRegExp

В библиотеке XRegExp имеется метод `matchChain()`, специально предназначенный для поиска совпадений с одним регулярным выражением внутри совпадений с другим регулярным выражением:

```
var result = XRegExp.matchChain(subject, [
    {regex: XRegExp("<b>(.*?)</b>", "s"), backref: 1},
    /\d+/
]);
```

Другое решение задачи, похожее на стандартное решение в JavaScript, основано на использовании метода `XRegExp.forEach()`:

```
var result = [];
var outerRegex = XRegExp("<b>(.*?)</b>", "s");
var innerRegex = /\d+/g;
XRegExp.forEach(subject, outerRegex, function(outerMatch) {
    var innerMatches = outerMatch[1].match(innerRegex);
    if (innerMatches) {
        result = result.concat(innerMatches);
    }
});
```

PHP

```
$list = array();
preg_match_all('%<b>(.*?)</b>%s', $subject, $outermatches,
               PREG_PATTERN_ORDER);
for ($i = 0; $i < count($outermatches[0]); $i++) {
    if (preg_match_all('/\d+/', $outermatches[1][$i], $innermatches,
                       PREG_PATTERN_ORDER)) {
        $list = array_merge($list, $innermatches[0]);
    }
}
```

Perl

```
while ($subject =~ m!<b>(.*?)</b>!gs) {
    push(@list, ($& =~ m/\d+/g));
}
```

Этот фрагмент будет работать, только если внутреннее регулярное выражение (‹\d+› в данном примере) не содержит сохраняющих групп, поэтому следует использовать несохраняющую группировку. Подробнее об этом рассказывается в рецепте 2.9.

Python

```
list = []
innerre = re.compile(r"\d+")
for outermatch in re.finditer("(?s)<b>(.*?)</b>", subject):
    list.extend(innerre.findall(outermatch.group(1)))
```

Ruby

```
list = []
subject.scan(/<b>(.*?)<\/b>/m) {|outergroups|
    list += outergroups[1].scan(/\d+/)
}
```

Обсуждение

Регулярные выражения прекрасно подходят для выделения лексем из текста, но они плохо подходят для парсинга текста. Под *выделением лексем* здесь понимается идентификация различных частей строки, таких как числа, слова, символы, теги, комментарии и пр. Регулярное выражение выполняет сканирование строки слева направо и пытается отыскать соответствие, опробуя разные альтернативы и различные количества символов. С такой работой регулярные выражения справляются просто прекрасно.

Под *парсингом* понимается процесс выявления взаимоотношений между лексемами. Например, в языках программирования комбинации лексем формируют инструкции, функции, классы, пространства имен и пр. Отслеживание назначения лексем внутри широкого контекста лучше реализовать в программном коде. В частности, регулярные выражения не в состоянии отслеживать нелинейный контекст, например вложенные конструкции.[1]

Поиск одной лексемы внутри другой лексемы часто пытаются реализовать с помощью регулярных выражений. Пара HTML-тегов отыскивается с помощью простого регулярного выражения ‹(.*?)›.[2] Числа

---

[1] В некоторых современных диалектах регулярных выражений была предпринята попытка ввести средства сбалансированного, или рекурсивного сопоставления. Однако применение этих средств приводит к настолько сложным регулярным выражениям, что это лишний раз подтверждает мое замечание о том, что парсинг лучше всего выполнять с помощью программного кода.

[2] Чтобы иметь возможность идентифицировать тег, располагающийся на нескольких строках, следует включить режим «точке соответствуют границы строк». Для JavaScript можно воспользоваться выражением ‹([\s\S]*?)›.

отыскиваются с помощью еще более простого регулярного выражения ‹\d+›. Но если попытаться объединить эти два выражения в одно, можно в результате получить нечто совсем иное:

```
\d+(?=(?:(?!<b>).)*</b>)
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Хотя это регулярное выражение решает задачу, поставленную в данном рецепте, понять его будет очень непросто. Даже знаток регулярных выражений должен будет тщательнейшим образом исследовать его, чтобы понять, что оно делает; или даже воспользоваться каким-либо инструментом, чтобы протестировать его. А ведь это лишь комбинация двух простых регулярных выражений.

Лучшее решение состоит в том, чтобы сохранить два регулярных выражения в первоначальном виде, а для их объединения использовать программный код. Получившийся при этом программный код будет немного длиннее, но намного проще для понимания, а простота программного кода является одной из основных причин, по которым используют регулярные выражения. Выражение, подобное ‹(.*?)›, легко поймет любой обладающий минимальным опытом работы с регулярными выражениями; к тому же такое выражение быстро сделает то, что в противном случае потребовало бы больше строк программного кода, более сложных в сопровождении.

Несмотря на то что решения в этом рецепте являются одними из самых сложных в этой главе, тем не менее они очевидны. Используются два регулярных выражения. «Внешнее» регулярное выражение применяется для поиска HTML-тегов и текста между ними, причем текст между тегами сохраняется первой сохраняющей группой. Работа с этим выражением реализована, как было показано в рецепте 3.11. Единственное отличие состоит в том, что комментарий, говорящий о том, где можно использовать совпадения, замещен программным кодом, позволяющим «внутреннему» регулярному выражению выполнить свою работу.

Второму регулярному выражению соответствуют цифры. Работа с этим выражением реализована, как было показано в рецепте 3.10. Единственное отличие состоит в том, что вместо целой строки второе регулярное выражение применяется только к части испытуемой строки, совпавшей с первой сохраняющей группой во внешнем регулярном выражении.

Существуют два способа ограничить внутреннее регулярное выражение текстом, совпавшим (с первой сохраняющей группой) во внешнем выражении. В некоторых языках программирования имеется функция, позволяющая применять регулярное выражение к части строки. Это может помочь сэкономить на копировании строк, если функция поиска соответствий не производит автоматическое заполнение структуры текстом, соответствующим сохраняющим группам. Всегда имеется

возможность извлечь подстроку, совпавшую с сохраняющей группой, и применить к ней внутреннее регулярное выражение.

В любом случае совместное использование двух регулярных выражений в цикле будет работать быстрее, чем использование одного регулярного выражения с вложенными опережающими проверками. Последнее требует, чтобы механизм регулярных выражений выполнял большое число возвратов. Для больших файлов единое регулярное выражение будет работать намного медленнее, так как ему потребуется определять границы раздела (HTML-теги `<b>`) для каждого числа, присутствующего в испытуемом тексте, включая числа, находящиеся вне тегов `<b>`. Решение, использующее два регулярных выражения, даже не пытается искать числа, пока не будут обнаружены границы раздела, время поиска которых зависит линейно от размера файла.

Библиотека XRegExp для JavaScript имеет особый метод `matchChain()`, специально предназначенный для поиска совпадений с одним регулярным выражением внутри совпадений с другим регулярным выражением. Этот метод принимает массив регулярных выражений во втором параметре. В массиве можно передать сколь угодно много регулярных выражений. Благодаря этому можно организовать поиск совпадений с регулярным выражением внутри совпадений с другим регулярным выражением, найденных внутри совпадений с третьим регулярным выражением, и т. д. В этом рецепте используется только два регулярных выражения, поэтому в массиве передается всего два элемента. Если потребуется, чтобы следующее выражение выполняло поиск внутри фрагментов, совпавших с определенной сохраняющей группой, добавьте это регулярное выражение в массив в виде объекта. Объект должен иметь свойство `regex` с регулярным выражением в нем и свойство `backref` с именем или номером сохраняющей группы. Если последнее регулярное выражение поместить в массив в виде объекта со свойствами `regex` и `backref`, возвращаемый массив будет содержать совпадения с указанной сохраняющей группой в финальном регулярном выражении.

См. также

В этом рецепте использовались приемы, представленные в трех рецептах выше. В рецепте 3.8 демонстрируется код, определяющий позицию и длину совпадения. В рецепте 3.10 демонстрируется код, извлекающий список всех совпадений, которые были найдены регулярным выражением в испытуемой строке. В рецепте 3.11 демонстрируется код, выполняющий обход всех совпадений, которые были найдены регулярным выражением в испытуемой строке.

3.14. Замена всех совпадений

Задача

Необходимо заменить все совпадения с регулярным выражением `‹before›` замещающим текстом `«after»`.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string resultString = Regex.Replace(subjectString, "before", "after");
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
string resultString = null;
try {
    resultString = Regex.Replace(subjectString, "before", "after");
} catch (ArgumentNullException ex) {
    // Не допускается передавать значение null в качестве регулярного
    // выражения, испытуемой строки или замещающего текста
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex("before");
string resultString = regexObj.Replace(subjectString, "after");
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект `Regex` с полной обработкой исключений:

```
string resultString = null;
try {
    Regex regexObj = new Regex("before");
    try {
        resultString = regexObj.Replace(subjectString, "after");
    } catch (ArgumentNullException ex) {
        // Не допускается передавать значение null в качестве регулярного
        // выражения или замещающего текста
    }
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim ResultString = Regex.Replace(SubjectString, "before", "after")
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
Dim ResultString As String = Nothing
Try
    ResultString = Regex.Replace(SubjectString, "before", "after")
Catch ex As ArgumentNullException
    'Не допускается передавать значение null в качестве регулярного
    'выражения, испытуемой строки или замещающего текста
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex**:**

```
Dim RegexObj As New Regex("before")
Dim ResultString = RegexObj.Replace(SubjectString, "after")
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex **с полной обработкой исключений:**

```
Dim ResultString As String = Nothing
Try
    Dim RegexObj As New Regex("before")
    Try
        ResultString = RegexObj.Replace(SubjectString, "after")
    Catch ex As ArgumentNullException
        'Не допускается передавать значение null в качестве регулярного
        'выражения или замещающего текста
    End Try
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Java

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
String resultString = subjectString.replaceAll("before", "after");
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
try {
    String resultString = subjectString.replaceAll("before", "after");
```

```
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
} catch (IllegalArgumentException ex) {
    // Синтаксическая ошибка в замещающем тексте
    // (неэкранированный символ $ ?)
} catch (IndexOutOfBoundsException ex) {
    // В замещающем тексте использована несуществующая обратная ссылка
}
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Matcher`**:**

```
Pattern regex = Pattern.compile("before");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("after");
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект `Matcher` **с полной обработкой исключений:**

```
String resultString = null;
try {
    Pattern regex = Pattern.compile("before");
    Matcher regexMatcher = regex.matcher(subjectString);
    try {
        resultString = regexMatcher.replaceAll("after");
    } catch (IllegalArgumentException ex) {
        // Синтаксическая ошибка в замещающем тексте
        // (неэкранированный символ $ ?)
    } catch (IndexOutOfBoundsException ex) {
        // В замещающем тексте использована несуществующая обратная ссылка
    }
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

JavaScript

```
result = subject.replace(/before/g, "after");
```

PHP

```
$result = preg_replace('/before/', 'after', $subject);
```

Perl

Испытуемая строка хранится в специальной переменной `$_`**, результат возвращается в переменной** `$_`**:**

```
s/before/after/g;
```

Испытуемая строка хранится в переменной `$subject`**, результат возвращается в переменной** `$subject`**:**

```
$subject =~ s/before/after/g;
```

Испытуемая строка хранится в переменной `$subject`, результат возвращается в переменной `$result`:

```
($result = $subject) =~ s/before/after/g;
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
result = re.sub("before", "after", subject)
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile("before")
result = reobj.sub("after", subject)
```

Ruby

```
result = subject.gsub(/before/, 'after')
```

Обсуждение

.NET

При реализации поиска с заменой с помощью регулярного выражения на платформе .NET всегда есть возможность использовать метод `Regex.Replace()`. Метод `Replace()` имеет 10 перегруженных версий. Половина из них принимает замещающий текст в виде строки; эти методы рассматриваются здесь. Другая половина принимает замещающий текст в виде делегата `MatchEvaluator`; эти методы рассматриваются в рецепте 3.16.

Первым аргументом, который ожидает получить метод `Replace()`, всегда является строка, хранящая оригинальный испытуемый текст, в котором требуется выполнить операцию поиска с заменой. Этот аргумент не может иметь значение `null`. В противном случае метод `Replace()` возбудит исключение `ArgumentNullException`. Метод `Replace()` всегда возвращает строку, в которой выполнены замены найденных совпадений.

Если регулярное выражение предполагается использовать всего несколько раз, можно применять статическую версию метода. Вторым аргументом этой версии метода передается используемое регулярное выражение. Замещающий текст передается в третьем аргументе. Параметры регулярного выражения можно передать в необязательном четвертом аргументе. Если регулярное выражение содержит синтаксическую ошибку, будет возбуждено исключение `ArgumentException`.

Если одно и то же регулярное выражение предполагается использовать для обработки большого числа строк, можно повысить эффективность программного кода, создав сначала объект `Regex`, а затем вызвав метод `Replace()` этого объекта. Первым аргументом этому методу передается

испытуемая строка, а вторым – замещающий текст. Это единственные обязательные аргументы.

При вызове метода `Replace()` относительно экземпляра класса `Regex` ему можно передать дополнительные аргументы, ограничивающие замещение совпадений. Если опустить эти аргументы, будут замещены все совпадения в испытуемой строке. Статическая версия метода `Replace()` не принимает эти дополнительные аргументы, она всегда замещает все совпадения.

В третьем необязательном аргументе, следующем за испытуемым и замещающим текстами, можно передать число выполняемых замен. Если число больше единицы, оно обозначает максимально возможное число замен. Например, вызов `Replace(subject, replacement, 3)` заменит только первые три совпадения с регулярным выражением, а все остальные будут игнорироваться. Если в строке будет найдено менее трех совпадений, замещены будут все совпадения. При этом нет никакой возможности определить ситуацию, когда число замен оказалось меньше, чем было запрошено. Если в третьем аргументе передать ноль, ни одной замены не будет произведено и будет возвращена строка в первоначальном виде. Если передать значение `-1`, будут замещены все совпадения с регулярным выражением. Если передать число меньше `-1`, метод `Replace()` возбудит исключение `ArgumentOutOfRangeException`.

Если методу был передан третий аргумент с числом замен, можно также передать четвертый необязательный аргумент, определяющий позицию символа, начиная с которого должно выполняться сопоставление с регулярным выражением. По сути, число, которое передается в четвертом аргументе, представляет собой количество символов в начале испытуемого текста, которые должны игнорироваться регулярным выражением. Этот аргумент может использоваться, когда строка уже обработана до некоторой позиции и необходимо выполнить поиск с заменой в оставшейся части строки. Значение четвертого аргумента должно быть числом в диапазоне от нуля до длины испытуемой строки. В противном случае метод `Replace()` возбудит исключение `ArgumentOutOfRangeException`. В отличие от метода `Match()`, `Replace()` не позволяет передавать аргумент, в котором можно было бы указать длину подстроки, где регулярному выражению позволено выполнять замены.

Java

Если требуется выполнить поиск с заменой в одной строке с помощью одного и того же регулярного выражения, можно воспользоваться методом `replaceFirst()` или `replaceAll()` данной строки. Оба метода принимают два аргумента: строку с регулярным выражением и строку с замещающим текстом. Эти функции удобно вызывать как `Pattern.compile("before").matcher(subjectString).replaceFirst("after")` и `Pattern.compile("before").matcher(subjectString).replaceAll("after")`.

Если одно и то же регулярное выражение предполагается использовать для обработки множества строк, следует создать объект `Matcher`, как описано в рецепте 3.3, а затем вызвать метод `replaceFirst()` или `replaceAll()` этого объекта, передавая ему замещающий текст в качестве единственного аргумента.

Если регулярное выражение и замещающий текст вводятся конечным пользователем, можно столкнуться с тремя классами исключений. Исключение `PatternSyntaxException` возбуждается методами `Pattern.compile()`, `String.replaceFirst()` и `String.replaceAll()`, если регулярное выражение содержит синтаксическую ошибку. Исключение `IllegalArgumentException` возбуждается методами `replaceFirst()` и `replaceAll()`, если синтаксическая ошибка присутствует в замещающем тексте. Если замещающий текст синтаксически оформлен правильно, но в нем имеются обратные ссылки на несуществующие сохраняющие группы, возбуждается исключение `IndexOutOfBoundsException`.

JavaScript

Чтобы выполнить поиск с заменой в строке при помощи регулярного выражения, можно использовать метод `replace()` этой строки. Первым аргументом этому методу передается регулярное выражение, а вторым – строка с замещающим текстом. Метод `replace()` возвращает новую строку с произведенными заменами.

Если необходимо выполнить замену всех совпадений с регулярным выражением, при создании объекта регулярного выражения следует установить флаг `/g`. Как это делается, описывается в рецепте 3.4. В отсутствие флага `/g` замещено будет только первое совпадение.

PHP

Чтобы выполнить поиск с заменой в строке при помощи регулярного выражения, можно использовать функцию `preg_replace()`. Регулярное выражение передается этой функции в первом аргументе, замещающий текст – во втором, а испытуемый текст – в третьем. Возвращаемое значение является строкой с испытуемым текстом, в котором выполнены замены.

С помощью четвертого необязательного аргумента можно ограничить число производимых замен. Если опустить этот аргумент или передать в нем значение `-1`, замещены будут все совпадения. Если указать значение `0`, ни одной замены произведено не будет. Если указать положительное число, функция `preg_replace()` произведет число замен, не превышающее указанного значения. Если число совпадений окажется меньше указанного числа, все имеющиеся совпадения будут замещены без появления ошибки.

Если потребуется узнать количество произведенных замен, можно передать функции пятый аргумент. В этот аргумент будет записано целое число фактически произведенных замен.

Особенность функции preg_replace() заключается в том, что в первых трех аргументах вместо строк она может принимать массивы. Если в третьем аргументе вместо единственной строки передается массив, функция preg_replace() вернет массив строк, произведя поиск с заменой во всех строках.

Если в первом аргументе передать массив строк с регулярными выражениями, функция preg_replace() будет использовать эти регулярные выражения одно за другим, выполняя операции поиска с заменой в испытуемом тексте. Если функции передается массив испытуемых строк, все регулярные выражения будут применены ко всем испытуемым строкам. Когда выполняется поиск с помощью массива регулярных выражений, в качестве замещающего текста можно указать единственную строку (которая будет использоваться всеми регулярными выражениями) или массив строк. При использовании двух массивов функция preg_replace() обойдет оба массива регулярных выражений и строк замещающего текста и будет использовать различные строки замещающего текста для каждого регулярного выражения. Обход строк в массивах будет выполняться в том порядке, в каком они хранятся в памяти, что не всегда совпадает с порядком следования индексов в массивах. Если элементы добавлялись в массивы не в порядке следования индексов, массивы регулярных выражений и замещающего текста следует обработать функцией ksort(), прежде чем передавать их функции preg_replace().

В следующем примере массив $replace создается в обратном порядке:

```
$regex[0] = '/a/';
$regex[1] = '/b/';
$regex[2] = '/c/';
$replace[2] = '3';
$replace[1] = '2';
$replace[0] = '1';

echo preg_replace($regex, $replace, "abc");
ksort($replace);
echo preg_replace($regex, $replace, "abc");
```

Первый вызов функции preg_replace() вернет строку 321, которая может оказаться не тем, что вы ожидали. После обработки массива функцией ksort() операция замены вернет строку 123, как и требовалось. Функция ksort() изменяет саму переменную, которая ей передается — не следует передавать ее возвращаемое значение (true или false) функции preg_replace().

Perl

В языке Perl конструкция s/// в действительности является оператором подстановки. Если оператор s/// используется самостоятельно, поиск с заменой будет выполняться в переменной $_, а результат будет помещаться обратно в переменную $_.

Если необходимо использовать другую переменную с оператором подстановки, следует применять оператор связывания =~, чтобы ассоциировать оператор подстановки с требуемой переменной. Связывание оператора подстановки со строкой приводит к немедленному выполнению операции поиска с заменой. Результат сохраняется в переменной, где хранился испытуемый текст.

Оператор s/// всегда изменяет переменную, с которой он связывается. Если необходимо сохранить результат поиска с заменой в другой переменной с целью оставить исходную переменную в неизменном виде, сначала нужно присвоить другой переменной содержимое исходной переменной, а затем связать оператор подстановки с этой переменной. Решение на языке Perl демонстрирует, как эти две операции можно выполнить в одной строке программного кода.

Чтобы выполнить замену всех совпадений с регулярным выражением, нужно использовать флаг /g, как описывается в рецепте 3.4. Без этого флага Perl заменит только первое совпадение.

Python

Для поиска с заменой в строке при помощи регулярного выражения можно использовать функцию sub() из модуля re. Регулярное выражение передается функции в первом аргументе, замещающий текст – во втором, а испытуемая строка – в третьем. Глобальная функция sub() не позволяет указывать параметры регулярного выражения.

Функция re.sub() вызывает функцию re.compile(), а затем метод sub() объекта скомпилированного регулярного выражения. Этот метод принимает два обязательных аргумента: замещающий текст и испытуемую строку.

Обе формы функции sub() возвращают строку, в которой замещены все совпадения с регулярным выражением. Обе принимают необязательный аргумент, который можно использовать для ограничения числа производимых замен. Если опустить этот аргумент или передать в нем нулевое значение, замещены будут все совпадения. Если передать положительное число, оно будет ограничивать максимальное число замещаемых совпадений. Если число совпадений окажется меньше указанного числа, все имеющиеся совпадения будут замещены без возникновения ошибки.

Ruby

Для поиска с заменой в строке при помощи регулярного выражения можно использовать метод gsub() класса String. Регулярное выражение передается методу в первом аргументе, а замещающий текст – во втором. Возвращаемым значением метода является строка с выполненными в ней заменами. Если в испытуемой строке совпадения с регулярным выражением отсутствуют, метод gsub() вернет оригинальную строку.

Метод `gsub()` не изменяет строку, относительно которой он вызывается. Если необходимо изменить оригинальную строку, следует вызывать метод `gsub!()`. Если в испытуемой строке совпадения с регулярным выражением отсутствуют, метод `gsub!()` вернет значение `nil`. В противном случае он вернет строку, относительно которой был произведен вызов, с выполненными в ней заменами.

См. также

Раздел «Поиск с заменой с помощью регулярных выражений» в главе 1, где описываются различные диалекты текста замены.

Рецепт 3.15, где демонстрируется код, выполняющий замену совпадений с повторным использованием фрагментов совпадений.

В рецепте 3.16 показано, как заменить все совпадения не фиксированными строками, а текстом, сгенерированным в программном коде.

3.15. Замена совпадений с повторным использованием частей совпадений

Задача

Необходимо выполнить поиск с заменой, при котором части совпадений с регулярным выражением будут вставляться обратно. Части, которые должны вставляться обратно, необходимо изолировать с помощью сохраняющих групп в регулярном выражении, как описывается в рецепте 2.9.

Например, необходимо отыскать пары слов, разделенных знаком равенства, и поменять эти слова местами.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string resultString = Regex.Replace(subjectString, @"(\w+)=(\w+)", "$2=$1");
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex(@"(\w+)=(\w+)");
string resultString = regexObj.Replace(subjectString, "$2=$1");
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim ResultString = Regex.Replace(SubjectString, "(\w+)=(\w+)", "$2=$1")
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("(\w+)=(\w+)")
Dim ResultString = RegexObj.Replace(SubjectString, "$2=$1")
```

Java

При обработке одной строки этим регулярным выражением можно использовать функцию `String.replaceAll()`**:**

```
String resultString = subjectString.replaceAll("(\\w+)=(\\w+)", "$2=$1");
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Matcher`**:**

```
Pattern regex = Pattern.compile("(\\w+)=(\\w+)");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("$2=$1");
```

JavaScript

```
result = subject.replace(/(\w+)=(\w+)/g, "$2=$1");
```

PHP

```
$result = preg_replace('/(\w+)=(\w+)/', '$2=$1', $subject);
```

Perl

```
$subject =~ s/(\w+)=(\w+)/$2=$1/g;
```

Python

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать глобальную функцию:

```
result = re.sub(r"(\w+)=(\w+)", r"\2=\1", subject)
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного выражения:

```
reobj = re.compile(r"(\w+)=(\w+)")
result = reobj.sub(r"\2=\1", subject)
```

Ruby

```
result = subject.gsub(/(\w+)=(\w+)/, '\2=\1')
```

Обсуждение

Регулярное выражение `‹(\w+)=(\w+)›` **совпадает с парой слов, и каждое из них сохраняется в своей собственной сохраняющей группе. Слово, стоящее перед знаком равенства, сохраняется в первой сохраняющей группе, второе слово, стоящее после знака равенства, сохраняется во второй сохраняющей группе.**

В замещающем тексте необходимо указать сначала текст, совпавший со второй сохраняющей группой, потом знак равенства, а затем текст, совпавший с первой сохраняющей группой. Сделать это можно с помощью специальных символов-заполнителей. В различных языках программирования синтаксис определения замещающего текста может изменяться в широких пределах. Диалекты замещающего текста рассматриваются в разделе «Поиск с заменой с помощью регулярных выражений» главы 1, а в рецепте 2.21 описывается, как ссылаться на сохраняющие группы в замещающем тексте.

.NET

В платформе .NET можно использовать метод `Regex.Replace()`, который был описан в предыдущем рецепте, используя в качестве замещающего текста строку. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста .NET, как описывается в рецепте 2.21.

Java

В языке Java можно использовать методы `replaceFirst()` и `replaceAll()`, которые рассматривались в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста Java, как описывается в этой книге.

JavaScript

В языке JavaScript можно использовать метод `string.replace()`, который был описан в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста JavaScript, как описывается в этой книге.

PHP

В языке PHP можно использовать функцию `preg_replace()`, которая рассматривалась в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста PHP, как описывается в этой книге.

Perl

В языке Perl часть `replace` в операторе `s/regex/replace/` интерпретируется как строка в кавычках. В замещающей строке допускается использовать специальные переменные `$&`, `$1`, `$2` и т. д., как описывается в рецептах 3.7 и 3.9. Значения этих переменных устанавливаются сразу же, как только будет найдено совпадение с регулярным выражением, непосредственно перед выполнением замены. Эти переменные можно использовать в любом другом программном коде на языке Perl. Их значения остаются неизменными, пока не будет предпринята другая попытка сопоставления с регулярным выражением.

Все остальные языки программирования, рассматриваемые в этой книге, предоставляют функцию, которая принимает замещающий текст в виде строки. Функция производит синтаксический анализ этой строки, чтобы обработать обратные ссылки, такие как $1 или \1. Но конструкция $1 не имеет в этих языках специального назначения за пределами замещающего текста.

Python

В языке Python можно использовать функцию sub(), которая рассматривалась в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста Python, как описывается в этой книге.

Ruby

В языке Ruby можно использовать метод String.gsub(), который рассматривался в предыдущем рецепте. Синтаксис добавления обратных ссылок в замещающий текст следует диалекту замещающего текста Ruby, как описывается в этой книге.

В языке Ruby не предоставляется возможность интерпретировать такие переменные, как $1, в замещающем тексте. Поэтому в Ruby интерпретация переменных выполняется перед вызовом gsub(). Но перед вызовом gsub() еще не найдено ни одно совпадение, поэтому невозможно выполнить подстановку обратных ссылок. Если попробовать интерпретировать переменную $1, будет получен текст совпадения с первой сохраняющей группой в последнем регулярном выражении, использовавшемся перед вызовом gsub().

Поэтому в качестве обратной ссылки в замещающем тексте следует использовать такие метасимволы, как «\1». Функция gsub() подставляет текст совпадения с соответствующей сохраняющей группой на место этих метасимволов для каждого найденного совпадения с регулярным выражением. Мы рекомендуем заключать замещающий текст в апострофы. В строках, окруженных кавычками, символ обратного слэша интерпретируется как экранирующий символ и экранирует восьмеричные цифры. Во фрагментах '\1' и "\\1" используется текст совпадения с первой сохраняющей группой, тогда как фрагмент "\1" будет замещен символом с кодом 0x01.

Именованное сохранение

При использовании в регулярном выражении именованных сохраняющих групп имеется возможность ссылаться из замещающего текста на группы по их именам.

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string resultString = Regex.Replace(subjectString,
                      @"(?<left>\w+)=(?<right>\w+)", "${right}=${left}");
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex**:**

```
Regex regexObj = new Regex(@"(?<left>\w+)=(?<right>\w+)");
string resultString = regexObj.Replace(subjectString, "${right}=${left}");
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim ResultString = Regex.Replace(SubjectString,
                   "(?<left>\w+)=(?<right>\w+)", "${right}=${left}")
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex**:**

```
Dim RegexObj As New Regex("(?<left>\w+)=(?<right>\w+)")
Dim ResultString = RegexObj.Replace(SubjectString, "${right}=${left}")
```

Java 7

В Java 7 появилась поддержка именованных сохраняющих групп в регулярных выражениях и именованных обратных ссылок в тексте замены.

При обработке одной строки этим регулярным выражением можно использовать функцию String.replaceAll()**:**

```
String resultString = subjectString.replaceAll(
                      "(?<left>\\w+)=(?<right>\\w+)", "${right}=${left}");
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Matcher**:**

```
Pattern regex = Pattern.compile("(?<left>\\w+)=(?<right>\\w+)");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("${right}=${left}");
```

XRegExp

Метод XRegExp.replace() **расширяет стандартный синтаксис определения текста замены в JavaScript возможностью использовать именованные обратные ссылки.**

```
var re = XRegExp("(?<left>\\w+)=(?<right>\\w+)", "g");
var result = XRegExp.replace(subject, re, "${right}=${left}");
```

PHP

```
$result = preg_replace('/(?P<left>\w+)=(?P<right>\w+)/',
                       '$2=$1', $subject);
```

Семейство функций `preg` в языке PHP использует библиотеку PCRE, которая поддерживает именованные сохранения. Функции `preg_match()` и `preg_match_all()` добавляют совпадения с именованными сохраняющими группами в массив с результатами. К сожалению, функция `preg_replace()` не предоставляет возможность использовать именованные обратные ссылки в замещающем тексте. Если регулярное выражение имеет именованные сохраняющие группы, нужно сосчитать именованные и нумерованные группы слева направо, чтобы определить номера обратных ссылок для каждой группы. Эти номера затем можно использовать в замещающем тексте.

Perl

```
$subject =~ s/(?<left>\w+)=(?<right>\w+)/$+{right}=$+{left}/g;
```

Именованные сохраняющие группы поддерживаются в Perl начиная с версии 5.10. В хеше %+ хранятся совпадения со всеми именованными сохраняющими группами, имеющимися в последнем использовавшемся регулярном выражении. Этот хеш можно использовать в строке с замещающим текстом, как и в любом другом месте.

Python

При обработке небольшого числа строк можно использовать глобальную функцию:

```
result = re.sub(r"(?P<left>\w+)=(?P<right>\w+)", r"\g<right>=\g<left>",
                subject)
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"(?P<left>\w+)=(?P<right>\w+)")
result = reobj.sub(r"\g<right>=\g<left>", subject)
```

Ruby

```
result = subject.gsub(/(?<left>\w+)=(?<right>\w+)/, '\k<left>=\k<right>')
```

См. также

Раздел «Поиск с заменой с помощью регулярных выражений» в главе 1, где описываются диалекты замещающего текста.

Рецепт 2.21, где объясняется, как добавлять обратные ссылки на сохраняющие группы в замещающий текст.

3.16. Замена совпадений фрагментами, сгенерированными в программном коде

Задача

Необходимо заменить все совпадения с регулярным выражением новым текстом, который генерируется программным кодом. При этом необходимо иметь возможность заменять все совпадения разными строками, основываясь на фактическом тексте совпадения.

Например, предположим, что необходимо заменить все числа в строке на их произведения с числом два.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string resultString = Regex.Replace(subjectString, @"\d+",
                      new MatchEvaluator(ComputeReplacement));
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex:

```
Regex regexObj = new Regex(@"\d+");
string resultString = regexObj.Replace(subjectString,
                      new MatchEvaluator(ComputeReplacement));
```

В обоих фрагментах вызывается функция ComputeReplacement**. Этот метод необходимо добавить в класс, реализующий это решение:**

```
public String ComputeReplacement(Match matchResult) {
    int twiceasmuch = int.Parse(matchResult.Value) * 2;
    return twiceasmuch.ToString();
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = Regex.Replace(SubjectString, "\d+", MyMatchEvaluator)
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex**:**

```
Dim RegexObj As New Regex("\d+")
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = RegexObj.Replace(SubjectString, MyMatchEvaluator)
```

В обоих фрагментах вызывается функция ComputeReplacement**. Этот метод необходимо добавить в класс, реализующий это решение:**

```
Public Function ComputeReplacement(ByVal MatchResult As Match) As String
    Dim TwiceAsMuch = Int.Parse(MatchResult.Value) * 2;
    Return TwiceAsMuch.ToString();
End Function
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    Integer twiceasmuch = Integer.parseInt(regexMatcher.group()) * 2;
    regexMatcher.appendReplacement(resultString, twiceasmuch.toString());
}
regexMatcher.appendTail(resultString);
```

JavaScript

```
var result = subject.replace(/\d+/g, function(match) {
    return match * 2; }
);
```

PHP

Используя объявленную ранее функцию обратного вызова:

```
$result = preg_replace_callback('/\d+/', 'compute_replacement', $subject);

function compute_replacement($groups) {
    return $groups[0] * 2;
}
```

Используя анонимную функцию обратного вызова:

```
$result = preg_replace_callback(
    '/\d+/',
    create_function(
        '$groups',
        'return $groups[0] * 2;'
    ),
    $subject
);
```

Perl

```
$subject =~ s/\d+/$& * 2/eg;
```

Python

При обработке небольшого числа строк можно использовать глобальную функцию:

```
result = re.sub(r"\d+", computereplacement, subject)
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile(r"\d+")
result = reobj.sub(computereplacement, subject)
```

В обоих фрагментах вызывается функция `computereplacement`. Эта функция должна быть объявлена до того, как она будет передана функции `sub()`.

```
def computereplacement(matchobj):
    return str(int(matchobj.group()) * 2)
```

Ruby

```
result = subject.gsub(/\d+/) {|match|
    Integer(match) * 2
}
```

Обсуждение

Когда в качестве замещающего текста используется строка, имеется возможность выполнять только простую подстановку текста. Чтобы каждое совпадение можно было заменить чем-то уникальным, основываясь на замещаемом совпадении, замещающий текст необходимо генерировать программным кодом.

C#

В рецепте 3.14 рассматривались различные способы вызова метода `Regex.Replace()`, когда ему в качестве замещающего текста передается строка. Статическая версия метода принимает замещающий текст в третьем аргументе, после испытуемого текста и регулярного выражения. Если передать регулярное выражение конструктору `Regex()`, появится возможность вызывать метод `Replace()` созданного объекта, который принимает замещающий текст во втором аргументе.

Вместо строки с замещающим текстом во втором или третьем аргументе допускается передавать делегат `MatchEvaluator`. Этот делегат является ссылкой на функцию-член, которая добавляется в класс, выполняющий операцию поиска с заменой. Создание делегата производится с помощью ключевого слова `new` и вызова конструктора `MatchEvaluator()`. Функция-член передается конструктору `MatchEvaluator()` в виде единственного аргумента.

Функция, которая будет использоваться для создания делегата, должна возвращать строку и принимать объект класса `System.Text.RegularExpressions.Match` в виде единственного аргумента. Это тот самый объект класса `Match`, который возвращается функцией-членом `Regex.Match()`, используемой почти во всех предыдущих рецептах этой главы.

Когда метод Replace() вызывается с делегатом MatchEvaluator в качестве замещающего текста, для каждого совпадения с регулярным выражением, которое требуется заменить, будет вызываться ваша функция. Эта функция должна возвращать замещающий текст. В процессе создания замещающего текста допускается использовать любые свойства объекта Match. В примере, показанном выше, для извлечения строки совпадения со всем выражением используется свойство matchResult.Value. На практике для сборки замещающего текста из совпадений с сохраняющими группами, имеющимися в регулярном выражении, часто используется коллекция matchResult.Groups[].

Если какие-то определенные совпадения не должны замещаться, функция должна возвращать значение свойства matchResult.Value. Если функция вернет значение null или пустую строку, совпадение с регулярным выражением будет замещено пустой строкой (т. е. будет удалено).

VB.NET

В рецепте 3.14 рассматривались различные способы вызова метода Regex.Replace(), когда ему в качестве замещающего текста передается строка. Статическая версия метода принимает замещающий текст в третьем аргументе, после испытуемого текста и регулярного выражения. Если использовать ключевое слово Dim для создания переменной с регулярным выражением, появится возможность вызывать метод Replace() этого объекта, который принимает замещающий текст во втором аргументе.

Вместо строки с замещающим текстом во втором или третьем аргументе допускается передавать объект класса MatchEvaluator. Этот объект содержит ссылку на функцию, которая добавляется в класс, выполняющий операцию поиска с заменой. Создание объекта класса MatchEvaluator производится с помощью ключевого слова Dim. Конструктору MatchEvaluator() передается ключевое слово AddressOf, за которым следует имя функции-члена. Оператор AddressOf возвращает ссылку на функцию, не производя фактический вызов этой функции.

Функция, которая будет использоваться в объекте MatchEvaluator, должна возвращать строку и принимать объект класса System.Text.RegularExpressions.Match в виде единственного аргумента. Это тот самый объект класса Match, который возвращается функцией-членом Regex.Match(), используемой почти во всех предыдущих рецептах этой главы. Аргумент передается функции по значению, поэтому он объявлен с ключевым словом ByVal.

Когда метод Replace() вызывается с объектом MatchEvaluator в качестве замещающего текста, для каждого совпадения с регулярным выражением, которое требуется заменить, будет вызываться ваша функция. Эта функция должна возвращать замещающий текст. В процессе создания замещающего текста допускается использовать любые свойства объекта Match. В примере, показанном выше, для извлечения строки совпадения со всем регулярным выражением используется свойство Match-

Result.Value. На практике для сборки замещающего текста из совпадений с сохраняющими группами, имеющимися в регулярном выражении, часто используется коллекция MatchResult.Groups[].

Если какие-то определенные совпадения не должны замещаться, функция должна возвращать значение свойства MatchResult.Value. Если функция вернет значение Nothing или пустую строку, совпадение с регулярным выражением будет замещено пустой строкой (т. е. будет удалено).

Java

Решение для Java выглядит очень просто. Здесь выполняются итерации через все совпадения с регулярным выражением, как описано в рецепте 3.11. Внутри цикла вызывается метод appendReplacement() объекта Matcher. Когда метод find() не может отыскать последующие совпадения, вызывается метод appendTail(). Применение методов appendReplacement() и appendTail() существенно упрощает возможность замены каждого совпадения уникальным замещающим текстом.

Метод appendReplacement() принимает два аргумента. Первый аргумент имеет тип StringBuffer и служит временным хранилищем результатов в процессе выполнения операции поиска с заменой. Второй аргумент – это замещающий текст для последнего совпадения, которое будет обнаружено методом find(). Этот замещающий текст может включать ссылки на сохраняющие группы, такие как "$1". Если в замещающем тексте имеется синтаксическая ошибка, будет возбуждено исключение IllegalArgumentException. Если в замещающем тексте имеется ссылка на несуществующую сохраняющую группу, будет возбуждено исключение IndexOutOfBoundsException. Если вызвать метод appendReplacement() после неудачной попытки отыскать совпадение функцией find(), будет возбуждено исключение IllegalStateException.

В случае успешного завершения вызова метода appendReplacement() выполняются два действия. Во-первых, он копирует в буфер текст, находящийся между предыдущим и текущим совпадениями, не выполняя никаких преобразований. Если текущее совпадение было первым, копируется весь текст, находящийся перед ним. Во-вторых, он добавляет замещающий текст, подставляя на место обратных ссылок совпадения с соответствующими сохраняющими группами.

Если требуется удалить какое-то совпадение, его достаточно просто заменить пустой строкой. Если необходимо оставить текст совпадения без изменения, нужно просто пропустить вызов метода appendReplacement() для этого совпадения. Под словами «предыдущее совпадение» подразумевается предыдущее совпадение, для которого вызывался метод appendReplacement(). Если для каких-то совпадений метод appendReplacement() не вызывался, они становятся частью текста между замещаемыми совпадениями, который копируется в буфер без изменения.

Завершив замещение совпадений, следует вызвать метод `appendTail()`. Он скопирует текст в конце строки, находящийся после последнего совпадения, для которого вызывался метод `appendReplacement()`.

JavaScript

В языке JavaScript функции являются обычными объектами, которые можно присваивать переменным. Вместо строкового литерала или переменной, которая хранит строку, в функцию `string.replace()` можно передать свою функцию, которая возвращает строку. Эта функция будет вызываться всякий раз, когда будет обнаруживаться необходимость выполнить замену.

Функцию, возвращающую замещающий текст, можно объявить так, что она будет принимать один или более аргументов. В этом случае в первом аргументе ей будет передаваться текст совпадения с регулярным выражением. Если в регулярном выражении имеются сохраняющие группы, во втором аргументе будет передан текст совпадения с первой сохраняющей группой, в третьем аргументе будет передан текст совпадения со второй сохраняющей группой и т. д. Благодаря этим аргументам можно использовать фрагменты совпадения с регулярным выражением для создания замещающего текста.

В решении на языке JavaScript для этого рецепта функция, возвращающая замещающий текст, просто принимает текст совпадения с регулярным выражением и возвращает число, умноженное на два. Преобразования из строки в число и обратно в языке JavaScript выполняются неявно.

PHP

Функция `preg_replace_callback()` действует точно так же, как и функция `preg_replace()`, которая была описана в рецепте 3.14. Она принимает регулярное выражение, замещение, испытуемый текст, необязательную начальную позицию и необязательное максимальное число замен. Регулярное выражение и испытуемый текст могут быть представлены обычными строками или массивами строк.

Отличие функции `preg_replace_callback()` состоит в том, что в качестве замещения (второй аргумент) она принимает не строку или массив строк, а функцию. Если эта функция объявлена в вашем коде, ее имя должно передаваться в виде строки. Как вариант можно создать анонимную функцию, передав результат вызова `create_function()`. В любом случае эта функция должна принимать один аргумент и возвращать строку (или что-то, что может быть преобразовано в строку).

Всякий раз когда функция `preg_replace_callback()` обнаруживает совпадение с регулярным выражением, она обращается к функции обратного вызова. Аргумент заполняется массивом строк. Нулевой элемент этого массива содержит совпадение со всем регулярным выражением, остальные элементы хранят совпадения с соответствующими сохраняющими

группами. Этот массив можно применять для сборки замещающего текста, используя текст совпадения с регулярным выражением или с одной или более сохраняющими группами.

Perl

Оператор s/// поддерживает один дополнительный модификатор, который игнорируется оператором m//: /e. Модификатор /e, или «execute» (выполнить), предписывает оператору подстановки выполнить часть оператора, содержащую замещающий текст, как программный код на языке Perl, а не интерпретировать ее как строку в кавычках. С помощью этого модификатора легко можно извлечь текст из переменной $& и умножить его на два. Результат работы программного кода используется как замещающий текст.

Python

Функция sub() в языке Python позволяет передавать функцию вместо строки с замещающим текстом. Эта функция будет вызываться для замещения каждого совпадения с регулярным выражением.

Функция должна объявляться до ссылки на нее и принимать один аргумент, экземпляр класса MatchObject, который является тем же самым объектом, что возвращает функция search(). Этот объект может использоваться для извлечения совпадений (или их частей) с регулярным выражением и конструирования замещающего текста на их основе. Подробнее об этом рассказывается в рецептах 3.7 и 3.9.

Функция обязана возвращать строку с замещающим текстом.

Ruby

В двух предыдущих рецептах использовался метод gsub() класса String с двумя аргументами: регулярное выражение и замещающий текст. Для этого метода существует блочная форма.

В блочной форме метод gsub() принимает единственный аргумент с регулярным выражением. Он присваивает первой переменной цикла строку с текстом совпадения для всего регулярного выражения. Если добавить дополнительные переменные цикла, они будут получать значение nil, даже если регулярное выражение содержит сохраняющие группы.

Внутрь блока следует поместить выражение, создающее строку, которая будет использоваться в качестве замещающего текста. Внутри блока допускается использовать специальные переменные, которые заполняются в процессе сопоставления с регулярным выражением, такие как $~, $& и $1. Подробнее об этом рассказывается в рецептах 3.7, 3.8 и 3.9.

В замещающем тексте не допускается использовать такие метасимволы, как «\1». Они будут интерпретироваться как обычный текст.

См. также

Рецепт 3.9, где демонстрируется код, извлекающий фрагмент испытуемой строки, совпавший с определенной частью регулярного выражения (сохраняющей группой).

Рецепт 3.15, где демонстрируется код, выполняющий замену совпадений с повторным использованием фрагментов совпадений.

3.17. Замена всех совпадений внутри совпадений с другим регулярным выражением

Задача

Необходимо заменить все совпадения с определенным регулярным выражением, но только внутри определенных участков испытуемого текста. Искомыми участками являются совпадения с другим регулярным выражением.

Допустим, что имеется файл HTML, в котором различные фрагменты выделены тегами . Внутри каждой пары тегов необходимо заменить все совпадения с регулярным выражением ‹before› замещающим текстом «after». Например, строка before first before before before before должна быть преобразована в строку before first after before after after.

Решение

C#

```
Regex outerRegex = new Regex("<b>.*?</b>", RegexOptions.Singleline);
Regex innerRegex = new Regex("before");
string resultString = outerRegex.Replace(subjectString,
                      new MatchEvaluator(ComputeReplacement));

public String ComputeReplacement(Match matchResult) {
    // Выполнить поиск с заменой
    // для каждого совпадения с «внешним» регулярным выражением
    return innerRegex.Replace(matchResult.Value, "after");
}
```

VB.NET

```
Dim OuterRegex As New Regex("<b>.*?</b>", RegexOptions.Singleline)
Dim InnerRegex As New Regex("before")
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = OuterRegex.Replace(SubjectString, MyMatchEvaluator)

Public Function ComputeReplacement(ByVal MatchResult As Match) As String
    'Выполнить поиск с заменой
    'для каждого совпадения с «внешним» регулярным выражением
```

```
    Return InnerRegex.Replace(MatchResult.Value, "after");
End Function
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern outerRegex = Pattern.compile("<b>.*?</b>");
Pattern innerRegex = Pattern.compile("before");
Matcher outerMatcher = outerRegex.matcher(subjectString);
while (outerMatcher.find()) {
    outerMatcher.appendReplacement(resultString,
        innerRegex.matcher(outerMatcher.group()).replaceAll("after"));
}
outerMatcher.appendTail(resultString);
```

JavaScript

```
var result = subject.replace(/<b>.*?<\/b>/g, function(match) {
    return match.replace(/before/g, "after");
});
```

PHP

```
$result = preg_replace_callback('%<b>.*?</b>%', replace_within_tag, $subject);
function replace_within_tag($groups) {
    return preg_replace('/before/', 'after', $groups[0]);
}
```

Perl

```
$subject =~ s%<b>.*?</b>%($match = $&) =~ s/before/after/g; $match;%eg;
```

Python

```
innerre = re.compile("before")
def replacewithin(matchobj):
    return innerre.sub("after", matchobj.group())

result = re.sub("<b>.*?</b>", replacewithin, subject)
```

Ruby

```
innerre = /before/
result = subject.gsub(/<b>.*?<\/b>/) {|match|
    match.gsub(innerre, 'after')
}
```

Обсуждение

Данное решение представляет собой комбинацию решений двух предыдущих рецептов и основано на использовании двух регулярных выражений. «Внешнему» регулярному выражению `<b>.*?</b>` соответствуют пары тегов `<b>` и текст между ними. «Внутреннему» регулярному выражению соответствует слово «before», которое замещается словом «after».

В рецепте 3.16 описывается, как выполнить поиск с заменой, конструируя замещающий текст для каждого совпадения с помощью программного кода. Здесь этот прием применяется для замещения совпадений с внешним регулярным выражением. Каждый раз когда внешнее регулярное выражение обнаруживает открывающий и закрывающий теги <b>, запускается операция поиска с заменой, как это делалось уже в рецепте 3.14. Испытуемым текстом для операции поиска с заменой во внутреннем регулярном выражении является текст, совпавший с внешним регулярным выражением.

См. также

В этом рецепте используется прием, описанный в трех предыдущих рецептах. Рецепт 3.11 демонстрирует код, выполняющий обход всех совпадений, которые были найдены регулярным выражением в испытуемой строке. Рецепт 3.13 демонстрирует код, выполняющий поиск совпадений внутри совпадений с другим регулярным выражением. Рецепт 3.16 демонстрирует возможность замены совпадений не фиксированными строками, а текстом, сгенерированным в программном коде.

3.18. Замена всех совпадений между совпадениями с другим регулярным выражением

Задача

Необходимо заменить все совпадения с некоторым регулярным выражением, но только внутри определенных участков испытуемого текста. Искомыми участками являются фрагменты, расположенные между совпадениями с другим регулярным выражением. Другими словами, необходимо выполнить поиск с заменой во всех фрагментах испытуемого текста, которые не совпали с другим регулярным выражением.

Предположим, что имеется файл HTML, в котором необходимо заменить все прямые кавычки ("") фигурными (“ ”), но только за пределами тегов HTML. Кавычки внутри тегов должны оставаться прямыми кавычками ASCII, в противном случае веб-браузер окажется не в состоянии выполнить парсинг разметки HTML. Например, строка "text" "text" "text" должна быть преобразована в “text” “text” “text”.

Решение

C#

```
string resultString = null;
Regex outerRegex = new Regex("<[^<>]*>");
```

```
Regex innerRegex = new Regex("\"([^\"]*)\"");
// Найти первый фрагмент
int lastIndex = 0;
Match outerMatch = outerRegex.Match(subjectString);
while (outerMatch.Success) {
    // Выполнить поиск с заменой в тексте
    // между этим и предыдущим совпадениями
    string textBetween =
        subjectString.Substring(lastIndex, outerMatch.Index - lastIndex);
    resultString += innerRegex.Replace(textBetween, "\u201C$1\u201D");
    lastIndex = outerMatch.Index + outerMatch.Length;
    // Скопировать текст фрагмента без изменений
    resultString += outerMatch.Value;
    // Найти следующий фрагмент
    outerMatch = outerMatch.NextMatch();
}
// Выполнить поиск с заменой в оставшейся части строки
// после последнего совпадения
string textAfter = subjectString.Substring(lastIndex,
                   subjectString.Length - lastIndex);
resultString += innerRegex.Replace(textAfter, "\u201C$1\u201D");
```

VB.NET

```
Dim ResultString As String = Nothing
Dim OuterRegex As New Regex("<[^<>]*>")
Dim InnerRegex As New Regex("""([^""]*)""")
'Найти первый фрагмент
Dim LastIndex = 0
Dim OuterMatch = OuterRegex.Match(SubjectString)
While OuterMatch.Success
    'Выполнить поиск с заменой в тексте
    'между этим и предыдущим совпадениями
    Dim TextBetween = SubjectString.Substring(LastIndex,
                      OuterMatch.Index - LastIndex);
    ResultString += InnerRegex.Replace(TextBetween,
                    ChrW(&H201C) + "$1" + ChrW(&H201D))
    LastIndex = OuterMatch.Index + OuterMatch.Length
    'Скопировать текст фрагмента без изменений
    ResultString += OuterMatch.Value
    'Найти следующий фрагмент
    OuterMatch = OuterMatch.NextMatch
End While
'Выполнить поиск с заменой в оставшейся части строки
'после последнего совпадения
Dim TextAfter = SubjectString.Substring(LastIndex,
                                        SubjectString.Length - LastIndex);
ResultString += InnerRegex.Replace(TextAfter,
                ChrW(&H201C) + "$1" + ChrW(&H201D))
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern outerRegex = Pattern.compile("<[^<>]*>");
Pattern innerRegex = Pattern.compile("\"([^\"]*)\"");
Matcher outerMatcher = outerRegex.matcher(subjectString);
int lastIndex = 0;
while (outerMatcher.find()) {
    // Выполнить поиск с заменой в тексте
    // между этим и предыдущим совпадениями
    String textBetween = subjectString.substring(lastIndex,
                                                 outerMatcher.start());
    Matcher innerMatcher = innerRegex.matcher(textBetween);
    resultString.append(innerMatcher.replaceAll("\u201C$1\u201D"));
    lastIndex = outerMatcher.end();
    // Добавить совпадение с регулярным выражением, не изменяя его
    resultString.append(outerMatcher.group());
}
// Выполнить поиск с заменой в оставшейся части строки
// после последнего совпадения
String textAfter = subjectString.substring(lastIndex);
Matcher innerMatcher = innerRegex.matcher(textAfter);
resultString.append(innerMatcher.replaceAll("\u201C$1\u201D"));
```

JavaScript

```
var result = "";
var outerRegex = /<[^<>]*>/g;
var innerRegex = /"([^"]*)"/g;
var outerMatch = null;
var lastIndex = 0;
while (outerMatch = outerRegex.exec(subject)) {
    if (outerMatch.index == outerRegex.lastIndex) outerRegex.lastIndex++;
    // Выполнить поиск с заменой в тексте между этим
    // и предыдущим совпадениями
    var textBetween = subject.slice(lastIndex, outerMatch.index);
    result += textBetween.replace(innerRegex, "\u201C$1\u201D");
    lastIndex = outerMatch.index + outerMatch[0].length;
    // Добавить совпадение с регулярным выражением, не изменяя его
    result += outerMatch[0];
}
// Выполнить поиск с заменой в оставшейся части строки
// после последнего совпадения
var textAfter = subject.slice(lastIndex);
result += textAfter.replace(innerRegex, "\u201C$1\u201D");
```

PHP

```
$result = '';
$lastindex = 0;
while (preg_match('/<[^<>]*>/', $subject, $groups, PREG_OFFSET_CAPTURE,
                  $lastindex)) {
```

```
$matchstart = $groups[0][1];
$matchlength = strlen($groups[0][0]);
// Выполнить поиск с заменой в тексте между этим
// и предыдущим совпадениями
$textbetween = substr($subject, $lastindex, $matchstart-$lastindex);
$result .= preg_replace('/"([^"]*)"/', '“$1”', $textbetween);
// Добавить совпадение с регулярным выражением, не изменяя его
$result .= $groups[0][0];
// Переместить начальную позицию для поиска следующего совпадения
$lastindex = $matchstart + $matchlength;
    if ($matchlength == 0) {
        // Устранить вероятность бесконечного цикла в случае,
        // когда регулярное выражение допускает возможность появления
        // совпадений нулевой длины
        $lastindex++;
    }
}
// Выполнить поиск с заменой в оставшейся части строки
// после последнего совпадения
$textafter = substr($subject, $lastindex);
$result .= preg_replace('/"([^"]*)"/', '“$1”', $textafter);
```

Perl

```
use encoding "utf-8";
$result = '';
while ($subject =~ m/<[^<>]*>/g) {
    $match = $&;
    $textafter = $';
    ($textbetween = $`) =~ s/"([^"]*)"/\x{201C}$1\x{201D}/g;
    $result .= $textbetween . $match;
}
$textafter =~ s/"([^"]*)"/\x{201C}$1\x{201D}/g;
$result .= $textafter;
```

Python

```
innerre = re.compile('"([^"]*)"')
result = "";
lastindex = 0;
for outermatch in re.finditer("<[^<>]*>", subject):
    # Выполнить поиск с заменой в тексте между этим
    # и предыдущим совпадениями
    textbetween = subject[lastindex:outermatch.start()]
    result += innerre.sub(u"\u201C\\1\u201D", textbetween)
    lastindex = outermatch.end()
    # Добавить совпадение с регулярным выражением, не изменяя его
    result += outermatch.group()
# Выполнить поиск с заменой в оставшейся части строки
# после последнего совпадения
textafter = subject[lastindex:]
result += innerre.sub(u"\u201C\\1\u201D", textafter)
```

Ruby

```
result = '';
textafter = ''
subject.scan(/<[^<>]*>/) {|match|
    textafter = $'
    textbetween = $`.gsub(/"([^"]*)"/, '“\1”')
    result += textbetween + match
}
result += textafter.gsub(/"([^"]*)"/, '“\1”')
```

Обсуждение

В рецепте 3.13 говорилось, как задействовать два регулярных выражения, чтобы отыскать совпадения (со вторым регулярным выражением) внутри определенных фрагментов в файле (совпавших с первым регулярным выражением). В решениях для этого рецепта используется тот же самый прием поиска и замены совпадений, только в пределах определенных фрагментов испытуемого текста.

Очень важно, чтобы регулярное выражение, которое используется для поиска фрагментов, разделяющих испытуемый текст на части, работало с оригинальной строкой. Если бы оригинальная строка модифицировалась, необходимо было бы смещать начальную позицию поиска для первого регулярного выражения, отыскивающего разделы, по мере добавления и удаления символов вторым регулярным выражением. Еще более важно, что модификации могут вносить нежелательные побочные эффекты. Например, если внешнее регулярное выражение использует якорный метасимвол ‹^›, чтобы привязать совпадение к началу строки, а внутреннее регулярное выражение вставляет символ перевода строки в конец раздела, найденного первым регулярным выражением, то метасимвол ‹^› будет совпадать сразу же за предыдущим разделом с вновь добавленным символом перевода строки.

Несмотря на то что решения получились довольно длинными, все они достаточно простые. В решениях используются два регулярных выражения. «Внешнему» регулярному выражению ‹<[^<>]*>› соответствуют пары угловых скобок и все, что находится между ними, за исключением самих угловых скобок. Это грубый способ поиска любых тегов HTML. Это регулярное выражение будет замечательно справляться со своей задачей при условии, что файл HTML не содержит каких-либо угловых скобок, которые не были (по ошибке) оформлены как сущности. Реализация работы с этим регулярным выражением в точности повторяет программный код из рецепта 3.11. Единственное отличие состоит в том, что комментарии, сообщающие, где можно использовать найденное совпадение, замещены программным кодом, который выполняет операцию поиска с заменой.

Поиск с заменой в цикле реализован точно так же, как решение для рецепта 3.14. Испытуемая строка, в которой выполняется поиск с заме-

ной, содержит фрагмент текста между предыдущим и текущим совпадениями с внешним регулярным выражением. Результат работы внутренней операции поиска с заменой добавляется в конец строки с общим результатом. Кроме того, в конец строки с общим результатом в неизменном виде добавляется текущее совпадение с внешним регулярным выражением.

Когда попытка отыскать совпадение с внешним регулярным выражением терпит неудачу, в последний раз запускается операция поиска с заменой в тексте, следующем за последним совпадением с внешним регулярным выражением.

Регулярному выражению ‹"([^"]*)"›, используемому внутри цикла для выполнения поиска с заменой, соответствуют пары кавычек и все, что находится между ними, за исключением самих кавычек. Текст между кавычками сохраняется первой сохраняющей группой.

В замещающем тексте используется ссылка на первую сохраняющую группу, которая помещена внутри пары фигурных кавычек. Фигурным кавычкам соответствуют пункты Юникода U+201C и U+201D. Обычно фигурные кавычки можно вставлять непосредственно в исходные тексты. Однако Visual Studio 2008, претендуя на интеллектуальность, автоматически замещает фигурные кавычки прямыми.

Внутри регулярного выражения сопоставление с кодовыми пунктами Юникода можно описать как ‹\u201C› или ‹\x{201C}›, но ни в одном из языков программирования, рассматриваемых в этой книге, не поддерживается возможность использовать такие конструкции в замещающем тексте. Если конечный пользователь пожелает вставить фигурные кавычки в замещающий текст, который он набирает в поле ввода, ему придется вставлять эти символы, копируя их из таблицы символов. В исходном тексте программы внутри замещающего текста можно использовать экранированные последовательности Юникода, если они поддерживаются языком программирования в строковых литералах. Например, языки C# и Java поддерживают конструкцию \u201C на уровне строки, но VB.NET не обеспечивает возможность добавления экранированных последовательностей Юникода в строки. В VB.NET для преобразования кодового пункта Юникода в символ можно воспользоваться функцией ChrW().

Perl и Ruby

В решениях на языках Perl и Ruby используются две специальные переменные, доступные в этих языках, о которых еще не рассказывалось. Переменная $` (знак доллара и обратный апостроф) хранит часть текста, расположенную слева от совпадения, а переменная $' (знак доллара и апостроф) хранит часть текста, расположенную справа от совпадения. Вместо того чтобы выполнять итерации по совпадениям в оригинальной строке с испытуемым текстом, мы начинаем новый поиск в части строки, следующей за предыдущим совпадением. При таком подходе

с помощью переменной $` легко можно извлечь текст, расположенный между предыдущим и текущим совпадениями.

Python

Результатом работы этого фрагмента кода является строка символов Юникода, потому что замещающий текст объявляется как строка Юникода. Чтобы ее отобразить, может потребоваться задействовать функцию encode(), как показано ниже:

```
print result.encode('1252')
```

См. также

В этом рецепте используется прием, описанный в трех предыдущих рецептах. Рецепт 3.11 демонстрирует код, выполняющий обход всех совпадений, которые были найдены регулярным выражением в испытуемой строке. Рецепт 3.13 демонстрирует код, выполняющий поиск совпадений внутри совпадений с другим регулярным выражением. Рецепт 3.16 демонстрирует возможность замены совпадений не фиксированными строками, а текстом, сгенерированным в программном коде.

3.19. Разбиение строки

Задача

Необходимо разбить строку с помощью регулярного выражения. После разбиения должен получиться массив строк, содержащих фрагменты текста между совпадениями с регулярным выражением. Например, требуется разбить строку, содержащую теги HTML, по этим тегам. Результатом разбиения строки `I•like•<b>bold</b>•and•<i>italic</i>•fonts` должен быть массив из пяти строк: `I•like•`, `bold`, `•and•`, `italic` и `•fonts`.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string[] splitArray = Regex.Split(subjectString, "<[^<>]*>");
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
string[] splitArray = null;
try {
    splitArray = Regex.Split(subjectString, "<[^<>]*>");
} catch (ArgumentNullException ex) {
    // Не допускается передавать значение null в качестве регулярного
    // выражения или испытуемой строки
```

```
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex**:**

```
Regex regexObj = new Regex("<[^<>]*>");
string[] splitArray = regexObj.Split(subjectString);
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex **с полной обработкой исключений:**

```
string[] splitArray = null;
try {
    Regex regexObj = new Regex("<[^<>]*>");
    try {
        splitArray = regexObj.Split(subjectString);
    } catch (ArgumentNullException ex) {
        // Не допускается передавать значение null в качестве регулярного
        // выражения или испытуемой строки
    }
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
```

Если регулярное выражение вводится конечным пользователем, следует использовать статическую функцию с полной обработкой исключений:

```
Dim SplitArray As String()
Try
    SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
Catch ex As ArgumentNullException
    'Не допускается передавать значение null в качестве регулярного
    'выражения или испытуемой строки
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект Regex**:**

```
Dim RegexObj As New Regex("<[^<>]*>")
Dim SplitArray = RegexObj.Split(SubjectString)
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект Regex **с полной обработкой исключений:**

```
Dim SplitArray As String()
Try
    Dim RegexObj As New Regex("<[^<>]*>")
    Try
        SplitArray = RegexObj.Split(SubjectString)
    Catch ex As ArgumentNullException
        'Не допускается передавать значение null в качестве регулярного
        'выражения или испытуемой строки
    End Try
Catch ex As ArgumentException
    'Синтаксическая ошибка в регулярном выражении
End Try
```

Java

Если необходимо разбить всего одну строку, используя это же регулярное выражение, можно использовать функцию `String.Split()`**:**

```
String[] splitArray = subjectString.split("<[^<>]*>");
```

Если регулярное выражение вводится конечным пользователем, следует предусмотреть полную обработку исключений:

```
try {
    String[] splitArray = subjectString.split("<[^<>]*>");
} catch (PatternSyntaxException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Pattern`**:**

```
Pattern regex = Pattern.compile("<[^<>]*>");
String[] splitArray = regex.split(subjectString);
```

Если регулярное выражение вводится конечным пользователем, следует использовать объект `Pattern` **с полной обработкой исключений:**

```
String[] splitArray = null;
try {
    Pattern regex = Pattern.compile("<[^<>]*>");
    splitArray = regex.split(subjectString);
} catch (ArgumentException ex) {
    // Синтаксическая ошибка в регулярном выражении
}
```

JavaScript

Разбивать строки с применением регулярного выражения можно с помощью метода *string*`.split()`**:**

```
result = subject.split(/<[^<>]*>/);
```

XRegExp

```
result = XRegExp.split(subject, /<[^<>]*>/);
```

PHP

```
$result = preg_split('/<[^<>]*>/', $subject);
```

Perl

```
@result = split(m/<[^<>]*>/, $subject);
```

Python

При разбиении небольшого числа строк можно использовать глобальную функцию:

```
result = re.split("<[^<>]*>", subject))
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile("<[^<>]*>")
result = reobj.split(subject)
```

Ruby

```
result = subject.split(/<[^<>]*>/)
```

Обсуждение

При разбиении строки с помощью регулярного выражения, по сути, решается задача, противоположная рецепту 3.10. Вместо получения списка всех совпадений с регулярным выражением в данном случае требуется получить список фрагментов текста между совпадениями, включая текст перед первым совпадением и после последнего совпадения. Сами совпадения с регулярным выражением исключаются из конечного результата.

C# и VB.NET

Чтобы разбить строку с помощью регулярного выражения, в платформе .NET всегда имеется возможность использовать метод `Regex.Split()`. Первым аргументом методу всегда передается строка, хранящая оригинальный испытуемый текст, который требуется разбить. В этом аргументе не допускается передавать значение `null`. В противном случае метод `Split()` возбудит исключение `ArgumentNullException`. Возвращаемым значением метода `Split()` всегда является массив строк.

Если регулярное выражение предполагается использовать всего несколько раз, можно использовать статическую версию метода. Вторым аргументом этой версии метода передается используемое регулярное выражение. Параметры регулярного выражения можно передавать в третьем необязательном аргументе. Если регулярное выражение содержит синтаксическую ошибку, будет возбуждено исключение `ArgumentException`.

Если одно и то же регулярное выражение предполагается использовать для обработки большого числа строк, можно повысить эффективность программного кода, создав сначала объект `Regex`, а затем вызывая метод `Split()` этого объекта. Единственным обязательным аргументом этого метода является испытуемая строка.

При вызове метода `Split()` экземпляра класса `Regex` ему можно передать дополнительные аргументы, ограничивающие количество разбиений. Если опустить эти аргументы, строка будет разбита по всем имеющимся совпадениям. Статическая версия метода `Split()` не принимает эти дополнительные аргументы, она всегда разбивает строку по всем имеющимся совпадениям.

Во втором необязательном аргументе, следующем за испытуемым текстом, можно передать максимальное число разбиений. Например, вызов `regexObj.Split(subject, 3)` вернет массив, содержащий не более трех строк. Метод `Split()` попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива.

Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, метод `Split()` выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного. Вызов `regexObj.Split(subject, 1)` вообще не выполнит ни одного разбиения и вернет массив с единственным элементом, в котором будет храниться оригинальная строка. Вызов `regexObj.Split(subject, 0)` выполнит разбиение по всем имеющимся совпадениям, как если бы метод `Split()` был вызван без второго аргумента. Если во втором аргументе передать отрицательное число, метод `Split()` возбудит исключение `ArgumentOutOfRangeException`.

Когда методу передается второй аргумент, определяющий максимальное число строк в возвращаемом массиве, можно также указать необязательный третий аргумент, определяющий позицию в испытуемой строке, с которой следует начинать поиск совпадений. По сути, число, которое передается в третьем аргументе, представляет собой число символов в начале испытуемого текста, которые должны игнорироваться регулярным выражением. Этот аргумент может использоваться, когда строка уже обработана до некоторой позиции и необходимо выполнить разбиение оставшейся части строки.

Символы, пропущенные регулярным выражением, тем не менее будут добавлены в возвращаемый массив. Первая строка в массиве – это вся подстрока, предшествующая первому обнаруженному после указанной начальной позиции совпадению, включая символы, находящиеся перед начальной позицией. Значением третьего аргумента должно быть число в диапазоне от нуля до длины испытуемой строки. В противном случае метод `Split()` возбудит исключение `ArgumentOutOfRangeException`.

В отличие от метода `Match()`, метод `Split()` не позволяет указывать длину подстроки, в которой можно выполнять поиск соответствий с регулярным выражением.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеются два совпадения, следующие непосредственно друг за другом без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку.

Java

Если требуется выполнить разбиение единственной строки, можно воспользоваться методом `split()` данной строки. Этот метод принимает регулярное выражение в виде единственного аргумента. Данный метод просто вызывает метод `Pattern.compile("regex").split(subjectString)`.

Если необходимо выполнить разбиение множества строк, следует создать объект `Pattern` с помощью фабричного метода `Pattern.compile()`. При таком подходе достаточно будет скомпилировать регулярное выражение всего один раз. После этого можно будет вызывать метод `split()` полученного экземпляра класса `Pattern`, передавая ему испытуемую строку в качестве аргумента. В данном случае нет необходимости создавать экземпляр класса `Matcher`. Класс `Matcher` вообще не имеет метода `split()`.

Метод `Pattern.split()` может принимать второй необязательный аргумент, а метод `String.split()` — нет. Во втором аргументе можно указать максимальное число фрагментов, которое требуется получить. Например, вызов `Pattern.split(subject, 3)` вернет массив, содержащий не более трех строк. Метод `split()` попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива. Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, метод `split()` выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного. Вызов `Pattern.split(subject, 1)` вообще не выполнит ни одного разбиения и вернет массив с единственным элементом, в котором будет храниться оригинальная строка.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеются два совпадения, следующие непосредственно друг за другом без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку.

Однако Java ликвидирует пустые строки в конце массива. Если необходимо, чтобы пустые строки включались в массив, методу `Pattern.split()` следует передать отрицательное число во втором аргументе. В этом случае разбиение будет выполнено по всем найденным совпадениям, и пустые строки в конце массива останутся на месте. Фактическая величина второго аргумента, когда в нем передается отрицательное число, не имеет значения. В языке Java невозможно одновременно ограничить число разбиений и оставить пустые строки в конце массива.

JavaScript

Чтобы выполнить разбиение строки в языке JavaScript, можно использовать метод `split()` этой строки. Чтобы получить массив строк, содержащий максимальное число разбиений, достаточно передать методу `split()` единственный аргумент с регулярным выражением. Во втором необязательном аргументе можно указать максимальное число строк в возвращаемом массиве. Это должно быть положительное число. Если в этом аргументе передать ноль, метод вернет пустой массив. Если опустить второй аргумент или передать в нем отрицательное число, строка будет разбита по всем найденным совпадениям. Наличие флага `/g` в регулярном выражении (рецепт 3.4) не оказывает никакого влияния.

В браузерах, совместимых со стандартами, метод `split()` включает в возвращаемый массив совпадения с сохраняющими группами. Более того, он даже добавляет значения `undefined` для сохраняющих групп, не участвовавших в сопоставлении. Если присутствие этих дополнительных элементов в возвращаемом массиве нежелательно, используйте только несохраняющие группы (рецепт 2.9) в регулярных выражениях, передаваемых методу `split()`.

Все основные веб-браузеры в настоящее время правильно реализуют метод `String.prototype.split()`. В старых браузерах имеются различные проблемы с сохраняющими группами и смежными совпадениями. Для тех, кому необходима реализация `String.prototype.split()`, следующая стандартам и работоспособная во всех браузерах, Стивен Левитан (Steven Levithan) предложил решение, доступное по адресу *http://blog.stevenlevithan.com/archives/cross-browser-split*.

XRegExp

При использовании библиотеки XRegExp в JavaScript, вызывайте `XRegExp.split(subject, regex)` вместо `subject.split(regex)`, чтобы получить стандартные результаты во всех браузерах.

PHP

Для разбиения строки на массив строк по совпадениям с регулярным выражением можно использовать функцию `preg_split()`. Регулярное выражение передается этой функции в первом аргументе, а испытуе-

мая строка – во втором. Если второй аргумент опущен, в качестве испытуемой строки используется специальная переменная $_.

В третьем необязательном аргументе можно указать максимальное число фрагментов, которое требуется получить. Например, вызов preg_split($regex, $subject, 3) вернет массив, содержащий не более трех строк. Функция preg_split() попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива. Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, функция preg_split() выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного. Если опустить третий аргумент или передать в нем значение -1, строка будет разбита по всем найденным совпадениям.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеются два совпадения, следующие непосредственно друг за другом без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку. По умолчанию функция preg_split() включает пустые строки в возвращаемый массив. Если наличие пустых строк в массиве нежелательно, можно передать константу PREG_SPLIT_NO_EMPTY в четвертом аргументе.

Perl

Для разбиения строки на массив строк по совпадениям с регулярным выражением можно использовать функцию split(). Оператор регулярного выражения передается этой функции в первом аргументе, а испытуемая строка – во втором.

В третьем необязательном аргументе можно указать максимальное число фрагментов, которое требуется получить. Например, вызов split(/regex/, subject, 3) вернет массив, содержащий не более трех строк. Функция split() попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива. Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, функция split() выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного.

Если опустить третий аргумент, интерпретатор Perl самостоятельно определит соответствующий предел. Если результат присваивается перемен-

ной-массиву, как это реализовано в решении к данному рецепту, строка будет разбита по всем найденным совпадениям. Если результат присваивается списку скалярных переменных, будет установлен предел, равный числу переменных в списке плюс один. Другими словами, будет выполнена попытка заполнить все переменные, а возможный остаток строки будет отброшен. Например, инструкция `($one, $two, $three) = split(/,/)` выполнит разбиение содержимого переменной `$_` на 4 части.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеются два совпадения, следующие непосредственно друг за другом без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если функции `split` передается третий необязательный параметр, имеющий достаточно большое значение или равный `-1`, и последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку, в противном случае пустая строка включаться в конец массива не будет.

Python

Чтобы разбить строку с помощью регулярного выражения, можно использовать функцию `split()` из модуля `re`. Регулярное выражение передается этой функции в первом аргументе, а испытуемая строка – во втором. Глобальная функция `split()` не позволяет передавать параметры регулярного выражения.

Функция `re.split()` вызывает функцию `re.compile()`, а затем метод `split()` объекта скомпилированного регулярного выражения. Этот метод имеет единственный обязательный аргумент – испытуемую строку.

Обе формы функции `split()` возвращают список с фрагментами текста между совпадениями с регулярным выражением. Обе принимают один необязательный аргумент, в котором можно указать максимальное число разбиений. Если опустить этот аргумент или передать в нем число ноль, строка будет разбита по всем найденным совпадениям. Если передать положительное число, оно будет определять максимальное количество совпадений с регулярным выражением, по которым следует разбить строку. Возвращаемый список будет содержать на одну строку больше, чем указано в аргументе. Последняя строка в списке будет представлять неразбитый остаток испытуемой строки после последнего совпадения. Если число найденных совпадений окажется меньше указанного значения, строка будет разбита по всем совпадениям без вывода сообщения об ошибке.

Ruby

Чтобы разбить испытуемую строку на массив строк по совпадениям с регулярным выражением, можно вызвать метод `split()` этой строки и передать ему регулярное выражение в первом аргументе.

Метод `split()` может принимать второй необязательный аргумент, в котором указывается максимальное число фрагментов, которое требуется получить. Например, вызов `subject.split(re, 3)` вернет массив, содержащий не более трех строк. Метод `split()` попытается отыскать два совпадения с регулярным выражением и вернет массив, содержащий текст перед первым совпадением, текст между двумя совпадениями и текст после второго совпадения. Все остальные возможные совпадения в строке будут игнорироваться и попадут в последнюю строку массива. Если число совпадений в строке окажется недостаточным, чтобы достигнуть указанного предела, метод `split()` выполнит разбиение по всем найденным совпадениям и вернет массив, количество строк в котором будет меньше указанного. Вызов `split(re, 1)` вообще не выполнит ни одного разбиения и вернет массив с единственным элементом, в котором будет храниться оригинальная строка.

Если первое совпадение будет обнаружено в начале испытуемой строки, первый элемент возвращаемого массива будет содержать пустую строку. Если в испытуемой строке имеются два совпадения, следующие непосредственно друг за другом без какого-либо текста между ними, в возвращаемый массив будет добавлена пустая строка. Если последнее совпадение будет обнаружено в конце испытуемой строки, последний элемент возвращаемого массива будет содержать пустую строку.

Однако Ruby ликвидирует пустые строки в конце массива. Если необходимо, чтобы пустые строки включались в массив, методу `split()` следует передать отрицательное число во втором аргументе. В этом случае разбиение будет выполнено по всем найденным совпадениям, и пустые строки в конце массива останутся на месте. Фактическая величина второго аргумента, когда в нем передается отрицательное число, не имеет значения. В языке Ruby невозможно одновременно ограничить число разбиений и оставить пустые строки в конце массива.

См. также

Рецепт 3.20, где демонстрируется код, разбивающий строку с использованием регулярного выражения и сохраняющий совпадения в массиве.

3.20. Разбиение строки, сохранение совпадений с регулярным выражением

Задача

Требуется выполнить разбиение строки с использованием регулярного выражения. В результате должен быть получен массив или список строк, содержащих фрагменты текста между совпадениями, а также с самими совпадениями.

Предположим, что требуется разбить строку с тегами HTML по этим тегам и дополнительно сохранить сами теги. Например, после разбиения

строки `I«like«<b>bold</b>«and«<i>italic</i>«fonts` должен получиться массив из девяти строк: `I«like«`, `<b>`, `bold`, `</b>`, `«and«`, `<i>`, `italic`, `</i>` и `«fonts`.

Решение

C#

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
string[] splitArray = Regex.Split(subjectString, "(<[^<>]*>)");
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Regex regexObj = new Regex("(<[^<>]*>)");
string[] splitArray = regexObj.Split(subjectString);
```

VB.NET

При обработке небольшого числа строк одним и тем же регулярным выражением можно использовать статическую функцию:

```
Dim SplitArray = Regex.Split(SubjectString, "(<[^<>]*>)")
```

В случае использования того же регулярного выражения для обработки большого числа строк предпочтительнее создать объект `Regex`:

```
Dim RegexObj As New Regex("(<[^<>]*>)")
Dim SplitArray = RegexObj.Split(SubjectString)
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("<[^<>]*>");
Matcher regexMatcher = regex.matcher(subjectString);
int lastIndex = 0;
while (regexMatcher.find()) {
    resultList.add(subjectString.substring(lastIndex, regexMatcher.start()));
    resultList.add(regexMatcher.group());
    lastIndex = regexMatcher.end();
}
resultList.add(subjectString.substring(lastIndex));
```

JavaScript

```
result = subject.split(/(<[^<>]*>)/);
```

XRegExp

```
result = XRegExp.split(subject, /(<[^<>]*>)/);
```

PHP

```
$result = preg_split('/(<[^<>]*>)/', $subject, -1, PREG_SPLIT_DELIM_CAPTURE);
```

Perl

```
@result = split(m/(<[^<>]*>)/, $subject);
```

Python

При разбиении небольшого числа строк можно использовать глобальную функцию:

```
result = re.split("(<[^<>]*>)", subject))
```

При многократном использовании одного и того же регулярного выражения предпочтительнее использовать объект скомпилированного регулярного выражения:

```
reobj = re.compile("(<[^<>]*>)")
result = reobj.split(subject)
```

Ruby

```
list = []
lastindex = 0;
subject.scan(/<[^<>]*>/) {|match|
    list << subject[lastindex..$~.begin(0)-1];
    list << $&
    lastindex = $~.end(0)
}
list << subject[lastindex..subject.length()]
```

Обсуждение

.NET

Метод `Regex.Split()` в платформе .NET включает в возвращаемый массив текст, совпавший с сохраняющими группами. В версиях .NET 1.0 и 1.1 в массив включается только совпадение с первой сохраняющей группой. В версиях .NET 2.0 и выше совпадения со всеми сохраняющими группами включаются в массив в виде отдельных строк. Если требуется включить в массив совпадение со всем регулярным выражением, его нужно заключить в общую сохраняющую группу. В версиях .NET 2.0 и выше необходимо, чтобы все остальные группы были несохраняющими, в противном случае совпадения с ними также попадут в массив.

Сохраняющие группы не участвуют в подсчете совпадений, когда методу `Split()` передается аргумент, определяющий максимальное число разбиений. Вызов `regexObj.Split(subject, 4)` с испытуемой строкой и регулярным выражением, которые приводятся в решении для данного рецепта, вернет массив, содержащий семь строк. Среди них будут четыре строки с фрагментами перед, между и после первых трех совпадений плюс три строки между ними, содержащие совпадения с регулярным выражением, которые были сохранены сохраняющими группами в регулярном выражении. Проще говоря, будет получен массив со строками: `I•like•`,

, bold, , «and», и italic«fonts. **Если бы регулярное выражение содержало 10 сохраняющих групп, вызов** regexObj.Split(subject, 4) **в версии .NET 2.0 или выше вернул бы массив с 34 строками.**

В .NET отсутствует возможность исключить совпадения с сохраняющими группами из массива. Единственное решение заключается в том, чтобы заменить все именованные и нумерованные сохраняющие группы на несохраняющие группы. Это легко можно сделать, передав параметр RegexOptions.ExplicitCapture **и заменив в регулярном выражении все именованные сохраняющие группы обычными (то есть группами в простых круглых скобках).**

Java

Метод Pattern.split() **в языке Java не обеспечивает возможность добавления совпадений с регулярным выражением в возвращаемый массив. Вместо этого можно адаптировать решение из рецепта 3.12, чтобы добавлять в список не только совпадения с регулярным выражением, но и фрагменты текста между совпадениями. Чтобы получить текст между совпадениями, в решении используется дополнительная информация о совпадениях, как описывается в рецепте 3.8.**

JavaScript

Функция *string*.split() **в языке JavaScript не предоставляет возможность управлять включением совпадений с регулярным выражением в массив. В соответствии со стандартом JavaScript в массив должны добавляться совпадения со всеми сохраняющими группами.**

В настоящее время все основные веб-браузеры правильно реализуют метод String.prototype.split(). **Старые версии браузеров не всегда корректно добавляли сохраняющие группы в возвращаемый массив. Для тех, кому необходима реализация** String.prototype.split(), **следующая стандартам и работоспособная во всех браузерах, Стивен Левитан (Steven Levithan) предложил решение, доступное по адресу *http://blog.stevenlevithan.com/archives/cross-browser-split*.**

XRegExp

При использовании библиотеки XRegExp в JavaScript вызывайте XRegExp.split(subject, regex) **вместо** subject.split(regex)**, чтобы получить стандартные результаты во всех браузерах.**

PHP

Если функции preg_split() **передать в четвертом аргументе значение** PREG_SPLIT_DELIM_CAPTURE**, в возвращаемый массив будут включены совпадения с сохраняющими группами. Значение** PREG_SPLIT_DELIM_CAPTURE **можно объединить со значением** PREG_SPLIT_NO_EMPTY **с помощью оператора |.**

Сохраняющие группы не участвуют в подсчете совпадений, когда функции `preg_split()` передается третий аргумент, определяющий максимальное число разбиений. Если установить предел числа разбиений, равный четырем, для испытуемой строки и регулярного выражения, которые приводятся в решении для данного рецепта, функция вернет массив, содержащий семь строк. Среди них будут четыре строки с фрагментами перед, между и после первых трех совпадений плюс три строки между ними, содержащие совпадения, которые были сохранены сохраняющими группами в регулярном выражении. Проще говоря, будет получен массив со строками: `I•like•`, `<b>`, `bold`, `</b>`, `•and•`, `<i>` и `italic</i>•fonts`.

Perl

Функция `split()` в языке Perl включает в массив совпадения со всеми сохраняющими группами. Если потребуется включить в массив совпадение со всем регулярным выражением, его нужно целиком заключить в сохраняющую группу.

Сохраняющие группы не участвуют в подсчете совпадений, когда функции `split()` передается аргумент, определяющий максимальное число разбиений. Вызов `split(/(<[^<>]*>)/, $subject, 4)` с испытуемой строкой и регулярным выражением, которые приводятся в решении для данного рецепта, вернет массив, содержащий семь строк. Среди них будут четыре строки с фрагментами перед, между и после первых трех совпадений плюс три строки между ними, содержащие совпадения с регулярным выражением, которые были сохранены сохраняющими группами в регулярном выражении. Проще говоря, будет получен массив со строками: `I•like•`, `<b>`, `bold`, `</b>`, `•and•`, `<i>` и `italic</i>•fonts`. Если бы регулярное выражение содержало 10 сохраняющих групп, вызов `split($regex, $subject, 4)` вернул бы массив с 34 строками.

В языке Perl отсутствует возможность исключить совпадения с сохраняющими группами из массива. Единственное решение заключается в том, чтобы заменить все именованные и нумерованные сохраняющие группы на несохраняющие группы.

Python

Функция `split()` в языке Python включает в массив совпадения со всеми сохраняющими группами. Если потребуется включить в массив совпадение со всем регулярным выражением, его нужно целиком заключить в сохраняющую группу.

Сохраняющие группы не участвуют в подсчете совпадений, когда функции `split()` передается аргумент, определяющий максимальное число разбиений. Вызов `split("(<[^<>]*>)", subject, 3)` с испытуемой строкой и регулярным выражением, которые приводятся в решении для данного рецепта, вернет массив, содержащий семь строк. Среди них будут четыре строки с фрагментами перед, между и после первых трех совпадений

плюс три строки между ними, содержащие совпадения с регулярным выражением, которые были сохранены сохраняющими группами в регулярном выражении. Проще говоря, будет получен массив со строками: `"I like"`, `"<b>"`, `"bold"`, `"</b>"`, `" and "`, `"<i>"` и `"italic</i> fonts"`. Если бы регулярное выражение содержало 10 сохраняющих групп, вызов `split(regex, subject, 3)` вернул бы массив с 34 строками.

В языке Python отсутствует возможность исключить совпадения с сохраняющими группами из массива. Единственное решение заключается в том, чтобы заменить все именованные и нумерованные сохраняющие группы на несохраняющие группы.

Ruby

Метод `String.split()` в языке Ruby не предоставляет возможность управлять включением в массив совпадений с регулярным выражением. Вместо этого можно адаптировать решение из рецепта 3.11, чтобы добавлять в список не только совпадения с регулярным выражением, но и фрагменты текста между совпадениями. Чтобы получить текст между совпадениями, в решении используется дополнительная информация о совпадениях, как описывается в рецепте 3.8.

См. также

Рецепт 2.9, где приводится описание сохраняющей и несохраняющей группировок.

Рецепт 2.11, где рассказывается об именованных сохраняющих группах. При разбиении строк некоторые языки программирования добавляют в массив также совпадения с сохраняющими группами.

Рецепт 3.19, где демонстрируется код, разбивающий строку без добавления в массив совпадений с регулярным выражением.

3.21. Построчный поиск

Задача

Традиционные инструменты grep применяют регулярное выражение к одной строке текста за один раз и отображают строки, совпавшие (или не совпавшие) с регулярным выражением. Допустим, что имеется некоторый массив строк или многострочный текст, который требуется обработать именно таким способом.

Решение

C#

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
string[] lines = Regex.Split(subjectString, "\r?\n");
```

Затем выполнить обход строк в массиве:

```
Regex regexObj = new Regex("regex pattern");
for (int i = 0; i < lines.Length; i++) {
    if (regexObj.IsMatch(lines[i])) {
        // Строка lines[i] соответствует регулярному выражению
    } else {
        // Строка lines[i] не соответствует регулярному выражению
    }
}
```

VB.NET

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
Dim Lines = Regex.Split(SubjectString, "\r?\n")
```

Затем выполнить обход строк в массиве:

```
Dim RegexObj As New Regex("regex pattern")
For i As Integer = 0 To Lines.Length - 1
    If RegexObj.IsMatch(Lines(i)) Then
        'Строка Lines[i] соответствует регулярному выражению
    Else
        'Строка Lines[i] не соответствует регулярному выражению
    End If
Next
```

Java

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
String[] lines = subjectString.split("\r?\n");
```

Затем выполнить обход строк в массиве:

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher("");
for (int i = 0; i < lines.length; i++) {
    regexMatcher.reset(lines[i]);
    if (regexMatcher.find()) {
        // Строка lines[i] соответствует регулярному выражению
    } else {
        // Строка lines[i] не соответствует регулярному выражению
    }
}
```

JavaScript

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку.

```
var lines = subject.split(/\r?\n/);
```

Затем выполнить обход строк в массиве:

```
var regexp = /regex pattern/;
for (var i = 0; i < lines.length; i++) {
    if (lines[i].match(regexp)) {
        // Строка lines[i] соответствует регулярному выражению
    } else {
        // Строка lines[i] не соответствует регулярному выражению
    }
}
```

PHP

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
$lines = preg_split('/\r?\n/', $subject)
```

Затем выполнить обход строк в массиве:

```
foreach ($lines as $line) {
    if (preg_match('/regex pattern/', $line)) {
        // Строка line соответствует регулярному выражению
    } else {
        // Строка line не соответствует регулярному выражению
    }
}
```

Perl

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
@lines = split(m/\r?\n/, $subject)
```

Затем выполнить обход строк в массиве:

```
foreach $line (@lines) {
    if ($line =~ m/regex pattern/) {
        # Строка $line соответствует регулярному выражению
    } else {
        # Строка $line не соответствует регулярному выражению
    }
}
```

Python

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
lines = re.split("\r?\n", subject);
```

Затем выполнить обход строк в массиве:

```
reobj = re.compile("regex pattern")
for line in lines[:]:
    if reobj.search(line):
        # Строка line соответствует регулярному выражению
    else:
        # Строка line соответствует регулярному выражению
```

Ruby

Если имеется многострочный текст, его сначала необходимо поместить в массив так, чтобы каждый элемент массива содержал единственную строку:

```
lines = subject.split(/\r?\n/)
```

Затем выполнить обход строк в массиве:

```
re = /regex pattern/
lines.each { |line|
    if line =~ re
        # Строка line соответствует регулярному выражению
    else
        # Строка line не соответствует регулярному выражению
}
```

Обсуждение

При необходимости выполнять построчную обработку данных можно обезопасить себя от большого числа возможных проблем, если разбить данные на отдельные строки, вместо того чтобы пытаться работать с одной большой строкой, содержащей символы перевода строки. После этого можно применять нужное регулярное выражение к каждой строке в массиве, не беспокоясь о том, что будет обнаружено совпадение одновременно с несколькими строками. Кроме того, такой подход упрощает отслеживание взаимоотношений между строками. Например, можно с помощью одного регулярного выражения отыскивать строки с заголовками, а с помощью другого – строки нижнего колонтитула. После этого при обнаружении разделяющих строк можно использовать третье регулярное выражение для поиска интересующих строк данных. На первый взгляд реализация такого алгоритма может показаться слишком трудоемкой, однако он достаточно простой, а программный код, реализующий его, обладает высокой эффективностью. Создать одно регулярное выражение, которое будет отыскивать заголовки, данные и нижние

колонтитулы, намного сложнее, и такое выражение будет работать намного медленнее.

Кроме того, при построчной обработке текста легко можно инвертировать регулярное выражение. Нет простого способа реализовать в регулярном выражении условие «соответствует строке, которая не содержит это или это слово». Легко инвертировать можно только символьные классы. Но когда текст уже разбит на строки, поиск строк, которые не содержат некоторое слово, становится таким же простым делом, как поиск текстового литерала во всех строках и удаление строк, в которых присутствует искомое слово.

В рецепте 3.19 показано, как легко можно разбить строку, превратив ее в массив строк. Регулярному выражению ‹\r\n› соответствуют пары символов CR и LF, которые разделяют строки в Microsoft Windows. Регулярному выражению ‹\n› соответствует символ LF, разделяющий строки в системе UNIX и ее производных, таких как Linux и даже OS X. Поскольку эти два регулярных выражения, по сути, являются простым текстом, для разбиения текста на строки можно даже вообще не использовать их. Если язык программирования позволяет разбивать строки по текстовым литералам, обязательно следует использовать эту возможность.

Если заранее точно неизвестно, как разделяются строки в используемых данных, можно выполнить разбиение текста с помощью регулярного выражения ‹\r?\n›. Так как здесь символ CR объявлен необязательным, это регулярное выражение будет соответствовать как CRLF, границам строк в Windows, так и LF, границам строк в UNIX.

Как только строки окажутся в массиве, их легко можно будет обойти в цикле. Внутри цикла можно будет определить, какие строки соответствуют регулярному выражению, а какие нет, следуя рецепту 3.5.

См. также

В этом рецепте используется прием, представленный в двух рецептах выше. Рецепт 3.11, где демонстрируется код, выполняющий обход всех совпадений, которые были найдены регулярным выражением в испытуемой строке. Рецепт 3.19, где демонстрируется код, с помощью регулярного выражения разбивающий строку и помещающий результат в массив или список.

3.22. Конструирование парсера

Задача

Имеется приложение, хранящее некоторые данные в таблице. Необходимо добавить в это приложение возможность импортировать данные из файлов, формат которых пока не поддерживается приложением.

Для данного формата не существует стандартных парсеров, поэтому необходимо реализовать собственный.

Новый формат файлов следует следующим правилам:

1. Новая таблица начинается с ключевого слова `table`. Файл может содержать любое число таблиц, но не менее одной.
2. Все строки, следующие за ключевым словом `table`, составляют заголовок таблицы. Однако таблицы могут не иметь заголовков.
3. Новая строка в таблице начинается с ключевого слова `row`. Строки не могут существовать за пределами таблиц. Таблица может содержать любое число строк, но не менее одной.
4. За ключевым словом `row` не может следовать простая текстовая строка.
5. Новая ячейка начинается с ключевого слова `cell`. Ячейка не может существовать за пределами строки таблицы. Строка таблицы может содержать любое число ячеек, в том числе и нулевое. Разные строки в одной таблице могут содержать разное число ячеек.
6. Любая текстовая строка, следующая за ключевым словом `cell`, является содержимым ячейки. Ячейка может не иметь содержимого.
7. Текстовая строка – это последовательность из нуля и более символов, заключенных между символами процента. Текстовая строка без каких-либо символов между символами процента – это пустая текстовая строка. Два идущих подряд символа процента в текстовой строке обозначают один символ процента. Никакие другие символы, отличные от символа процента, не имеют специального значения в текстовых строках. Разрывы строк и другие управляющие символы, находящиеся между символами процента, являются частью текстовой строки.
8. Если за ключевым словом `table` или `cell` следуют две или более текстовых строк, они образуют отдельные текстовые строки в заголовке таблицы или в содержимом ячейки, независимо от того, имеются ли символы разрыва строк между ними в исходном файле.
9. Ключевые слова не чувствительны к регистру. Все слова – `Cell`, `cell`, `CELL` и `CeLl` – интерпретируются как ключевое слово `cell`.
10. Любые пробельные символы между ключевыми словами и/или строками должны игнорироваться. Пробельные символы обязательно должны использоваться для отделения друг от друга соседних ключевых слов. Пробельные символы обязательно должны использоваться для отделения соседних текстовых строк. Пробельные символы не требуются для отделения ключевых слов от текстовых строк.
11. Любые символы, не являющиеся частью ключевого слова или текстовой строки, должны интерпретироваться как ошибка.

Ниже приводится пример содержимого файла, иллюстрирующего эти правила:

```
table %First table%
  row cell %A1% cell %B1% cell%C1%cell%D1%
  ROW row CELL %Предыдущая строка пуста%
  cell %B3%
  row
    cell %A4% %вторая строка%
    cEll %B4%
         %вторая строка%
    cell %C4
вторая строка%
row cell %%%текст%%%
  cell %%
  cell %%%%
  cell %%%%%%
```

которое может быть представлено в табличной форме, как показано в табл. 3.1.

Таблица 3.1. Таблица, которая должна получиться из файла примера

A1	B1	C1	D1
(не определена)	(не определена)	(не определена)	(не определена)
Предыдущая строка пуста	B3	(не определена)	(не определена)
A4	B4	C4	(не определена)
вторая строка %текст%	вторая строка (пустая)	вторая строка %	%%

Решение должно определять функцию, выполняющую парсинг содержимого файла, целиком хранящегося в одной строковой переменной. Функция в обязательном порядке должна использовать имеющиеся в приложении структуры `RECTable`, `RECRow` и `RECCell`, предназначенные для хранения таблиц, импортированных из файлов.

Решение

C#

```
static RECTable ImportTable(string fileContents) {
RECTable table = null;
RECRow row = null;
RECCell cell = null;
Regex regexObj = new Regex(
    @" \b(?<keyword>table|row|cell)\b
    | %(?<string>[^%]*(?:%%[^%]*)*)%
    | (?<error>\S+)",
  RegexOptions.IgnoreCase | RegexOptions.IgnorePatternWhitespace);
Match match = regexObj.Match(fileContents);
while (match.Success) {
```

```
if (match.Groups["keyword"].Success) {
  string keyword = match.Groups["keyword"].Value.ToLower();
  if (keyword == "table") {
    table = new RECTable();
    row = null;
    cell = null;
  } else if (keyword == "row") {
    if (table == null)
      throw new Exception("Invalid data: row without table");
    row = table.addRow();
    cell = null;
  } else if (keyword == "cell") {
    if (row == null)
      throw new Exception("Invalid data: cell without row");
    cell = row.addCell();
  } else {
    throw new Exception("Parser bug: unknown keyword");
  }
} else if (match.Groups["string"].Success) {
  string content = match.Groups["string"].Value.Replace("%%", "%");
  if (cell != null)
    cell.addContent(content);
  else if (row != null)
    throw new Exception("Invalid data: string after row keyword");
  else if (table != null)
    table.addCaption(content);
  else
    throw new Exception("Invalid data: string before table keyword");
  } else if (match.Groups["error"].Success) {
    throw new Exception("Invalid data: " + match.Groups["error"].Value);
  } else {
    throw new Exception("Parser bug: no capturing group matched");
  }
   match = match.NextMatch();
 }
 if (table == null)
   throw new Exception("Invalid data: table keyword missing");
 return table;
}
```

VB.NET

```
Function ImportTable(ByVal FileContents As String)
  Dim Table As RECTable = Nothing
  Dim Row As RECRow = Nothing
  Dim Cell As RECCell = Nothing
  Dim RegexObj As New Regex(
    " \b(?<keyword>table|row|cell)\b" & _
    "| %(?<string>[^%]*(?:%%[^%]*)*)%" & _
    "| (?<error>\S+)",
    RegexOptions.IgnoreCase Or RegexOptions.IgnorePatternWhitespace)
  Dim MatchResults As Match = RegexObj.Match(FileContents)
```

```
While MatchResults.Success
  If MatchResults.Groups("keyword").Success Then
    Dim Keyword As String = MatchResults.Groups("keyword").Value
    Keyword = Keyword.ToLower()
    If Keyword = "table" Then
      Table = New RECTable
      Row = Nothing
      Cell = Nothing
    ElseIf Keyword = "row" Then
      If Table Is Nothing Then
        Throw New Exception("Invalid data: row without table")
      End If
      Row = Table.addRow
      Cell = Nothing
    ElseIf Keyword = "cell" Then
      If Row Is Nothing Then
        Throw New Exception("Invalid data: cell without row")
      End If
      Cell = Row.addCell
    Else
      Throw New Exception("Parser bug: unknown keyword")
    End If
  ElseIf MatchResults.Groups("string").Success Then
    Dim Content As String = MatchResults.Groups("string").Value
    Content = Content.Replace("%%", "%")
    If Cell IsNot Nothing Then
      Cell.addContent(Content)
    ElseIf Row IsNot Nothing Then
      Throw New Exception("Invalid data: string after row keyword")
    ElseIf Table IsNot Nothing Then
      Table.addCaption(Content)
    Else
      Throw New Exception("Invalid data: string before table keyword")
    End If
  ElseIf MatchResults.Groups("error").Success Then
    Throw New Exception("Invalid data")
  Else
    Throw New Exception("Parser bug: no capturing group matched")
  End If
  MatchResults = MatchResults.NextMatch()
  End While
  If Table Is Nothing Then
    Throw New Exception("Invalid data: table keyword missing")
  End If
  Return Table
End Function
```

Java

```
RECTable ImportTable(String fileContents) throws Exception {
  RECTable table = null;
  RECRow row = null;
```

```
RECCell cell = null;
final int groupkeyword = 1;
final int groupstring = 2;
final int grouperror = 3;
Pattern regex = Pattern.compile(
    " \\b(table|row|cell)\\b\n" +
    "| %([^%]*(?:%%[^%]*)*)%\n" +
    "| (\\S+)",
    Pattern.CASE_INSENSITIVE | Pattern.COMMENTS);
Matcher regexMatcher = regex.matcher(fileContents);
while (regexMatcher.find()) {
  if (regexMatcher.start(groupkeyword) >= 0) {
    String keyword = regexMatcher.group(groupkeyword).toLowerCase();
    if (keyword.equals("table")) {
      table = new RECTable();
      row = null;
      cell = null;
    } else if (keyword.equals("row")) {
      if (table == null)
        throw new Exception("Invalid data: row without table");
      row = table.addRow();
      cell = null;
    } else if (keyword.equals("cell")) {
      if (row == null)
        throw new Exception("Invalid data: cell without row");
      cell = row.addCell();
    } else {
      throw new Exception("Parser bug: unknown keyword");
    }
  } else if (regexMatcher.start(groupstring) >= 0) {
    String content = regexMatcher.group(groupstring);
    content = content.replaceAll("%%", "%");
    if (cell != null)
      cell.addContent(content);
    else if (row != null)
      throw new Exception("Invalid data: String after row keyword");
    else if (table != null)
      table.addCaption(content);
    else
      throw new Exception("Invalid data: String before table keyword");
  } else if (regexMatcher.start(grouperror) >= 0) {
      throw new Exception("Invalid data: " +
                          regexMatcher.group(grouperror));
    } else {
      throw new Exception("Parser bug: no capturing group matched");
    }
  }
  if (table == null)
    throw new Exception("Invalid data: table keyword missing");
  return table;
}
```

JavaScript

```
function importTable(fileContents) {
  var table = null;
  var row = null;
  var cell = null;
  var groupkeyword = 1;
  var groupstring = 2;
  var grouperror = 3;
  var myregexp = /\b(table|row|cell)\b|%([^%]*(?:%%[^%]*)*)%|(\S+)/ig;
  var match;
  var keyword;
  var content;
  while (match = myregexp.exec(fileContents)) {
    if (match[groupkeyword] !== undefined) {
      keyword = match[groupkeyword].toLowerCase();
      if (keyword == "table") {
        table = new RECTable();
        row = null;
        cell = null;
      } else if (keyword == "row") {
        if (!table)
          throw new Error("Invalid data: row without table");
        row = table.addRow();
        cell = null;
      } else if (keyword == "cell") {
        if (!row)
          throw new Error("Invalid data: cell without row");
        cell = row.addCell();
      } else {
        throw new Error("Parser bug: unknown keyword");
      }
    } else if (match[groupstring] !== undefined) {
      content = match[groupstring].replace(/%%/g, "%");
      if (cell)
        cell.addContent(content);
      else if (row)
        throw new Error("Invalid data: string after row keyword");
      else if (table)
        table.addCaption(content);
      else
        throw new Error("Invalid data: string before table keyword");
      } else if (match[grouperror] !== undefined) {
        throw new Error("Invalid data: " + match[grouperror]);
      } else {
        throw new Error("Parser bug: no capturing group matched");
      }
    }
    if (!table)
      throw new Error("Invalid data: table keyword missing");
    return table;
}
```

XRegExp

```
function importTable(fileContents) {
  var table = null;
  var row = null;
  var cell = null;
  var myregexp = XRegExp("(?ix)\\b(?<keyword>table|row|cell)\\b" +
                         "    | %(?<string>[^%]*(?:%%[^%]*)*)%" +
                         "    | (?<error>\\S+)");
  XRegExp.forEach(fileContents, myregexp, function(match) {
    var keyword;
    var content;
    if (match.keyword !== undefined) {
      keyword = match.keyword.toLowerCase();
      if (keyword == "table") {
        table = new RECTable();
        row = null;
        cell = null;
      } else if (keyword == "row") {
        if (!table)
          throw new Error("Invalid data: row without table");
        row = table.addRow();
        cell = null;
      } else if (keyword == "cell") {
        if (!row)
          throw new Error("Invalid data: cell without row");
        cell = row.addCell();
      } else {
        throw new Error("Parser bug: unknown keyword");
      }
    } else if (match.string !== undefined) {
      content = match.string.replace(/%%/g, "%");
      if (cell)
        cell.addContent(content);
      else if (row)
        throw new Error("Invalid data: string after row keyword");
      else if (table)
        table.addCaption(content);
      else
        throw new Error("Invalid data: string before table keyword");
    } else if (match.error !== undefined) {
        throw new Error("Invalid data: " + match.error);
      } else {
        throw new Error("Parser bug: no capturing group matched");
      }
    });
    if (!table)
      throw new Error("Invalid data: table keyword missing");
    return table;
  }
```

Perl

```
sub importtable {
  my $filecontents = shift;
  my $table;
  my $row;
  my $cell;
  while ($filecontents =~
         m/ \b(table|row|cell)\b
         | %([^%]*(?:%%[^%]*)*)%
         | (\S+)/ixg) {
    if (defined($1)) { # Ключевое слово
      my $keyword = lc($1);
      if ($keyword eq "table") {
        $table = new RECTable();
        undef $row;
        undef $cell;
      } elsif ($keyword eq "row") {
        if (!defined($table)) {
          die "Invalid data: row without table";
        }
        $row = $table->addRow();
        undef $cell;
      } elsif ($keyword eq "cell") {
        if (!defined($row)) {
          die "Invalid data: cell without row";
        }
        $cell = $row->addCell();
      } else {
        die "Parser bug: unknown keyword";
      }
    } elsif (defined($2)) { # Текстовая строка
      my $content = $2;
      $content =~ s/%%/%/g;
      if (defined($cell)) {
        $cell->addContent($content);
      } elsif (defined($row)) {
        die "Invalid data: string after row keyword";
      } elsif (defined($table)) {
        $table->addCaption($content);
      } else {
        die "Invalid data: string before table keyword";
      }
    } elsif (defined($3)) { # Ошибка
      die "Invalid data: $3";
    } else {
      die "Parser bug: no capturing group matched";
    }
  }
  if (!defined($table)) {
    die "Invalid data: table keyword missing";
  }
```

```
  return $table;
}
```

Python

```
def importtable(filecontents):
  table = None
  row = None
  cell = None
  for match in re.finditer(
r"""(?ix)\b(?P<keyword>table|row|cell)\b
        | %(?P<string>[^%]*(?:%%[^%]*)*)%
        | (?P<error>\S+)""", filecontents):
if match.group("keyword") != None:
  keyword = match.group("keyword").lower()
  if keyword == "table":
    table = RECTable()
    row = None
    cell = None
  elif keyword == "row":
    if table == None:
      raise Exception("Invalid data: row without table")
    row = table.addRow()
    cell = None
  elif keyword == "cell":
    if row == None:
      raise Exception("Invalid data: cell without row")
    cell = row.addCell()
  else:
    raise Exception("Parser bug: unknown keyword")
elif match.group("string") != None:
  content = match.group("string").replace("%%", "%")
  if cell != None:
    cell.addContent(content)
  elif row != None:
    raise Exception("Invalid data: string after row keyword")
  elif table != None:
    table.addCaption(content)
  else:
    raise Exception("Invalid data: string before table keyword")
elif match.group("error") != None:
    raise Exception("Invalid data: " + match.group("error"))
  else:
    raise Exception("Parser bug: no capturing group matched")
if table == None:
  raise Exception("Invalid data: table keyword missing")
return table
```

PHP

```
function importTable($fileContents) {
  preg_match_all(
    '/ \b(?P<keyword>table|row|cell)\b
```

```
    | (?P<string>%[^%]*(?:%%[^%]*)*%)
    | (?P<error>\S+)/ix',
  $fileContents, $matches, PREG_PATTERN_ORDER);
$table = NULL;
$row = NULL;
$cell = NULL;
for ($i = 0; $i < count($matches[0]); $i++) {
  if ($matches['keyword'][$i] != NULL) {
    $keyword = strtolower($matches['keyword'][$i]);
    if ($keyword == "table") {
      $table = new RECTable();
      $row = NULL;
      $cell = NULL;
    } elseif ($keyword == "row") {
      if ($table == NULL)
        throw new Exception("Invalid data: row without table");
      $row = $table->addRow();
      $cell = NULL;
    } elseif ($keyword == "cell") {
      if ($row == NULL)
        throw new Exception("Invalid data: cell without row");
      $cell = $row->addCell();
    } else {
      throw new Exception("Parser bug: unknown keyword");
    }
  } elseif ($matches['string'][$i] != NULL) {
    $content = $matches['string'][$i];
    $content = substr($content, 1, strlen($content)-2);
    $content = str_replace('%%', '%', $content);
    if ($cell != NULL)
      $cell->addContent($content);
    elseif ($row != NULL)
      throw new Exception("Invalid data: string after row keyword");
    elseif ($table != NULL)
      $table->addCaption($content);
    else
      throw new Exception("Invalid data: string before table keyword");
  } elseif ($matches['error'][$i] != NULL) {
    throw new Exception("Invalid data: " + $matches['error'][$i]);
  } else {
      throw new Exception("Parser bug: no capturing group matched");
    }
  }
  if ($table == NULL)
    throw new Exception("Invalid data: table keyword missing");
  return $table;
}
```

Ruby

```
def importtable(filecontents)
  table = nil
```

```
row = nil
cell = nil
groupkeyword = 0;
groupstring = 1;
grouperror = 2;
regexp = / \b(table|row|cell)\b
          | %([^%]*(?:%%[^%]*)*)%
          | (\S+)/ix
filecontents.scan(regexp) do |match|
  if match[groupkeyword]
    keyword = match[groupkeyword].downcase
    if keyword == "table"
      table = RECTable.new()
      row = nil
      cell = nil
    elsif keyword == "row"
      if table.nil?
        raise "Invalid data: row without table"
      end
      row = table.addRow()
      cell = nil
    elsif keyword == "cell"
      if row.nil?
        raise "Invalid data: cell without row"
      end
      cell = row.addCell()
    else
      raise "Parser bug: unknown keyword"
    end
  elsif not match[groupstring].nil?
    content = match[groupstring].gsub("%%", "%")
    if not cell.nil?
      cell.addContent(content)
    elsif not row.nil?
      raise "Invalid data: string after row keyword"
    elsif not table.nil?
      table.addCaption(content)
    else
      raise "Invalid data: string before table keyword"
    end
  elsif not match[grouperror].nil?
      raise "Invalid data: " + match.group("error")
    else
      raise "Parser bug: no capturing group matched"
    end
  end
  if table.nil?
    raise "Invalid data: table keyword missing"
  end
  return table
end
```

Обсуждение

Самый простой способ реализовать парсер – выделять лексемы с помощью регулярного выражения и анализировать их в программном коде.

Под *выделением лексем* понимается поиск в файле лексем, которые являются наименьшими синтаксическими элементами. В соответствии с описанием формата файла лексемами являются: три ключевых слова, текстовые строки, заключенные в символы процента, пробельные символы между ключевыми словами и текстовыми строками и непробельные символы за пределами ключевых слов и текстовых строк. Создать регулярное выражение, соответствующее каждой из этих лексем, не составляет никакого труда.

```
\b(?<keyword>table|row|cell)\b
| %(?<string>[^%]*(?:%%[^%]*)*)%
| (?<error>\S+)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?P<keyword>table|row|cell)\b
| %(?P<string>[^%]*(?:%%[^%]*)*)%
| (?P<error>\S+)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и выше, Perl 5.10, Python

```
\b(table|row|cell)\b
| %([^%]*(?:%%[^%]*)*)%
| (\S+)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
\b(table|row|cell)\b|%([^%]*+(?:%%[^%]*+)*+)%|(\S+)
```

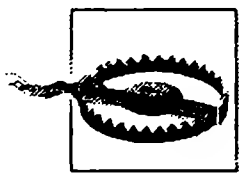
Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если в цикле применить данное регулярное выражение к содержимому файла примера, оно выделит каждое ключевое слово и каждую текстовую строку. Если применить его к файлу, содержащему недопустимые символы, каждая последовательность недопустимых символов будет выделена в отдельное совпадение. Регулярное выражение не совпадает с пробельными символами между ключевыми словами и текстовыми строками, потому что парсеру не требуется обрабатывать их. Метасимволы границ слов, окружающие список ключевых слов, – это все, что необходимо, чтобы выделить из содержимого файла ключевые слова, разделенные пробельными символами. Для поиска каждого типа лексем

используется отдельная сохраняющая группа; это упрощает их идентификацию в программном коде.

Чтобы сделать программный код более удобочитаемым там, где диалекты регулярных выражений поддерживают режим свободного форматирования и сохраняющую группировку, используются режимы свободного форматирования и нечувствительности к регистру символов. Между представленными регулярными выражениями нет никаких функциональных отличий.

Сохраняющая группа, совпадающая с текстовыми строками, не включает окружающие их символы процента. Благодаря этому в программном коде не приходится удалять эти символы, чтобы получить содержимое совпавших строк. Однако в этом есть один недостаток – когда регулярное выражение сопоставляется с пустой текстовой строкой (два символа процента, следующих подряд), сохраняющая группа обнаруживает совпадение нулевой длины. Поэтому при анализе совпадения с сохраняющей группой необходимо интерпретировать совпадение с нулевой длиной как допустимое. Например, в решении на языке JavaScript для этого используется выражение `if (match[groupstring] !== undefined)`, которое интерпретируется как истинное, если группа нашла совпадение, даже если совпадение имеет нулевую длину. Здесь нельзя использовать выражение `if (match[groupstring])`, потому что для нулевого совпадения оно вернет `false`.



Версии Internet Explorer 8 и ниже не следуют стандарту JavaScript, требующему, чтобы группы, не участвовавшие в сопоставлении, возвращали значение `undefined`. В IE8 такие группы хранят пустые строки, что не позволяет отличать их от групп, участвовавших в сопоставлении и нашедших совпадения нулевой длины. Это означает, что решение на JavaScript не будет работать в версиях IE8 и ниже. Эта ошибка была исправлена в Internet Explorer 9.

Метод `XRegExp.exec()` возвращает объект с результатами сопоставления, в котором группы, не участвовавшие в сопоставлении, имеют значение `undefined` независимо от версии браузера. То же относится и к методу `XRegExp.forEach()`, который основан на методе `XRegExp.exec()`. Если вам потребуется получить решение для браузеров, таких как IE8, которые не соответствуют стандартам в этом отношении, используйте решение на основе библиотеки XRegExp.

В PHP функция `preg_match_all()` сохраняет в массиве значение `NULL` и для сохраняющих групп, обнаруживших совпадение нулевой длины, и для сохраняющих групп, не участвовавших в сопоставлении. По этой причине решение на языке PHP включает символы процента в совпадение с группой `string`, из-за чего понадобилось добавить строку программного кода с вызовом функции `substr()`, удаляющей их.

Сам парсер реализован в виде программного кода. Парсер имеет четыре разных состояния. По своему состоянию он определяет, какой переменной – `table`, `row` или `cell` – должно быть присвоено значение.

1. Пустое состояние: еще ничего не было прочитано. Все переменные – `table`, `row` и `cell` – остаются неинициализированными.
2. Внутри таблицы: было извлечено ключевое слово `table`. Инициализируется переменная `table`, переменные `row` и `cell` остаются неинициализированными. Так как таблица может иметь заголовок, состоящий из произвольного числа текстовых строк, включая нулевое, парсеру не требуется переходить в отдельное состояние, чтобы начать анализ текстовых строк после ключевого слова `table`.
3. Внутри строки: было извлечено ключевое слово `row`. Инициализируется переменная `row`, при этом переменная `table` уже должна быть инициализирована, а `cell` – нет.
4. Внутри ячейки: было извлечено ключевое слово `cell`. Все переменные – `table`, `row` и `cell` – инициализированы. Так как ячейка может иметь содержимое, состоящее из произвольного числа текстовых строк, включая нулевое, парсеру не требуется переходить в отдельное состояние, чтобы начать анализ текстовых строк после ключевого слова `cell`.

При вызове парсера он начинает выполнять итерации по всем совпадениям с регулярным выражением, проверяет тип найденной лексемы – ключевое слово (`keyword`), текстовая строка (`string`) или недопустимый текст (`error`) – и обрабатывает эту лексему в зависимости от текущего состояния, как показано в табл. 3.2.

Таблица 3.2. Порядок обработки лексем в зависимости от состояния парсера

Совпадение	Состояние			
	Пустое	В таблице	В строке	В ячейке
Ключевое слово `table`	Создать новую таблицу и изменить состояние на «внутри таблицы»	Создать новую таблицу и изменить состояние на «внутри таблицы»	Создать новую таблицу и изменить состояние на «внутри таблицы»	Создать новую таблицу и изменить состояние на «внутри таблицы»
Ключевое слово `row`	Ошибка: недопустимые данные	Добавить строку в таблицу и изменить состояние на «внутри строки»	Добавить строку в таблицу	Добавить строку в таблицу и изменить состояние на «внутри строки»

Совпадение	Состояние			
	Пустое	В таблице	В строке	В ячейке
Ключевое слово `cell`	Ошибка: недопустимые данные	Ошибка: недопустимые данные	Добавить ячейку в строку и изменить состояние на «внутри ячейки»	Добавить ячейку в строку
Текстовая строка	Ошибка: недопустимые данные	Добавить строку в заголовок таблицы	Ошибка: недопустимые данные	Добавить содержимое в ячейку
Недопустимый текст	Ошибка: недопустимые данные	Ошибка: недопустимые данные	Ошибка: недопустимые данные	Ошибка: недопустимые данные

См. также

Приемы, использовавшиеся при составлении регулярного выражения в данном рецепте, обсуждаются в главе 2. В рецепте 2.6 рассказывается о границах слов, а в рецепте 2.8 – об операторе выбора, использовавшемся для сопоставления с ключевыми словами. В рецепте 2.11 описываются именованные сохраняющие группы. Применение именованных групп упрощает чтение и сопровождение регулярных выражений.

Для выделения текстовых строк, заключенных в символы процента, используется прием, описываемый в рецепте 7.8, реализующем поиск строк в кавычках в исходном программном коде. Единственное отличие – использование символа процента вместо кавычек.

Парсер выполняет итерации по всем совпадениям, найденным регулярным выражением. В рецепте 3.11 описывается, как действует этот механизм.

4

Проверка и форматирование

В этой главе содержатся рецепты реализации проверки и форматирования наиболее типичных видов данных, которые вводятся пользователем. Некоторые решения демонстрируют, как выполнить проверку данных, допускающих альтернативные форматы представления, такие как почтовые индексы США, которые могут состоять из пяти или девяти цифр. Другие предназначены для преобразования введенных данных, таких как номера телефонов, даты и номера кредитных карт, в наиболее общеупотребимые формы представления.

Эти рецепты помогут вам не только отклонять неправильный ввод, но и повысить удобство ваших приложений для пользователей. Например, сообщение «Вводить без дефисов и пробелов» рядом с полем ввода номера телефона или номера кредитной карты нередко игнорируется пользователями. К счастью, во многих случаях регулярные выражения помогут обеспечить пользователям возможность вводить данные в удобных и привычных для них форматах и не потребуют больших усилий с вашей стороны.

В некоторых языках программирования имеются функциональные возможности, похожие на те, что предлагаются в ряде рецептов этой главы, и реализованные через их собственные классы или библиотеки. В конкретных ситуациях бывает предпочтительнее использовать эти встроенные возможности, поэтому мы будем указывать на них по ходу обсуждения.

4.1. Проверка адресов электронной почты

Задача

Имеется форма на веб-сайте или диалог в приложении, где у пользователя запрашивается адрес электронной почты. Необходимо с помощью

регулярного выражения проверить корректность адреса, прежде чем пытаться отправить сообщение по этому адресу. Это позволит сократить число возвратов недоставленных сообщений.

Решение

Простое

Первое решение выполняет очень простую проверку. Оно проверяет лишь наличие символа @ в середине и отсутствие пробельных символов в адресе:

```
^\S+@\S+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A\S+@\S+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Простая проверка с ограничением допустимого набора символов

Доменное имя, часть адреса, следующая за символом @, может содержать только те символы, которые допускается использовать в доменных именах. Интернационализированные доменные имена не допускаются. *Локальный элемент*, часть адреса, предшествующая символу @, может содержать только те символы, которые допускается использовать в локальных элементах адресов электронной почты, количество которых намного меньше, чем могут принимать большинство клиентов и серверов электронной почты:

```
^[A-Z0-9+_.-]+@[A-Z0-9.-]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[A-Z0-9+_.-]+@[A-Z0-9.-]+\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Простая проверка без ограничения допустимого набора символов в локальном элементе адреса

Данное регулярное выражение является расширенной версией предыдущего и считает допустимым применять в именах пользователей более широкий набор символов. Не все программы электронной почты в состоянии обрабатывать все эти символы, но мы включили все символы, которые считаются допустимыми в соответствии с документом RFC 5322, в котором определяется формат сообщений электронной почты. Среди допустимых символов имеются такие, которые представляют

угрозу, например апострофы (') и символы вертикальной черты (|). Символы, которые могут иметь специальное назначение, обязательно необходимо экранировать при передаче адресов электронной почты другим программам, чтобы предотвратить такие виды нападений, как инъекция кода SQL:

```
^[A-Z0-9_!#$%&'*+/=?`{|}~^.-]+@[A-Z0-9.-]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[A-Z0-9_!#$%&'*+/=?`{|}~^.-]+@[A-Z0-9.-]+\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Без ведущих, завершающих или следующих друг за другом точек

В локальных элементах и в доменных именах допускается использовать один или более символов точки, но две точки, следующие подряд, считаются недопустимыми. Кроме того, точки не могут быть первыми и последними символами в имени пользователя и в доменном имени:

```
^[A-Z0-9_!#$%&'*+/=?`{|}~^-]+(?:\.[A-Z0-9_!#$%&'*+/=?`{|}~^-]+↵
)*@[A-Z0-9-]+(?:\.[A-Z0-9-]+)*$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[A-Z0-9_!#$%&'*+/=?`{|}~^-]+(?:\.[A-Z0-9_!#$%&'*+/=?`{|}~^-]+↵
)*@[A-Z0-9-]+(?:\.[A-Z0-9-]+)*\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Имя домена верхнего уровня может содержать от двух до шести букв

Это регулярное выражение добавляет к предыдущим версиям следующее требование к доменному имени: оно должно включать как минимум одну точку и после последней точки могут следовать только алфавитные символы. То есть имя домена должно состоять как минимум из двух уровней, например `secondlevel.com` или `thirdlevel.secondlevel.com`. Имя домена верхнего уровня (в данных примерах это `.com`) должно содержать от двух до шести алфавитных символов. Все коды стран, используемые в качестве домена верхнего уровня (`.us`, `.uk` и другие), состоят из двух букв. Имена универсальных доменов верхнего уровня обычно содержат от трех (`.com`) до шести букв (`.museum`):

```
^[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↵
(?:[A-Z0-9-]+\.)+[A-Z]{2,6}$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↵
(?:[A-Z0-9-]+\.)+[A-Z]{2,6}\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Об адресах электронной почты

Если на ваш взгляд для чего-то концептуально простого, такого как проверка адресов электронной почты, должно существовать простое решение, охватывающее все возможные вариации, то вы глубоко не правы. Этот рецепт представляет собой яркий пример ситуации, когда прежде чем браться за создание регулярного выражения, необходимо точно решить, что именно требуется проверить. Не существует универсального правила, согласно которому можно было бы однозначно сказать, какой адрес является корректным, а какой – нет. Все зависит от определения, *что считать правильным*.

Согласно документу RFC 5322, в котором определяется синтаксис адресов электронной почты, адрес `asdf@asdf.asdf` является корректным. Но он не будет считаться правильным, если под правильным понимать адрес, который примет сообщение электронной почты. В мире не существует домена верхнего уровня `asdf`.

Главная проблема проверки состоит в том, что невозможно точно утверждать, что адрес `john.doe@somewhere.com` правильный и способен принимать сообщения электронной почты, не попытавшись отправить письмо по этому адресу. Но даже после этого нельзя понять, что означает отсутствие ответа: домен `somewhere.com` уничтожил послание на несуществующий почтовый ящик, не сообщая об ошибке, или John Doe нажал клавишу Delete на клавиатуре, или письмо было уничтожено спам-фильтром.

Поскольку в любом случае придется проверять существование конкретного адреса посылкой сообщения на этот адрес, можно ограничиться более простым и менее строгим регулярным выражением. Принимать некорректные адреса может оказаться предпочтительнее, чем вызывать недовольство пользователей, отвергая корректные адреса. Учитывая это обстоятельство, можно отдать предпочтение регулярному выражению, выполняющему «простую проверку без ограничения допустимого набора символов в локальном элементе адреса». Хотя оно и пропускает то, что не является адресом электронной почты, например `#$%@.-`, тем не менее регулярное выражение само по себе очень простое и быстрое, и оно никогда не заблокирует допустимые адреса.

Если надо избежать слишком большого числа возвратов недоставленных писем и при этом не блокировать действительные адреса электронной почты, регулярное выражение «имя домена верхнего уровня может содержать от двух до шести букв» будет лучшим выбором.

Сложность регулярного выражения также необходимо принимать во внимание. Если выполняется проверка ввода пользователя, предпочтение стоит отдать более сложному регулярному выражению, потому что пользователь может ввести все что угодно. Но если адреса извлекаются из файлов базы данных, в отношении которых заранее известно, что они содержат корректные адреса, можно использовать очень простое регулярное выражение, которое просто отделяет адреса электронной почты от остальных данных. В этом случае достаточно будет даже регулярного выражения из раздела «Простое».

Наконец, необходимо учитывать пригодность регулярного выражения в будущем. В прошлом можно было ограничить имя домена верхнего уровня двухбуквенными кодами стран и перечислением универсальных доменов, например ‹com|net|org|mil|edu›. Но так как постоянно появляются новые домены, такие регулярные выражения быстро становятся непригодными.

Синтаксис регулярного выражения

Регулярные выражения, представленные в этом рецепте, демонстрируют все основные элементы синтаксиса регулярных выражений в действии. Если вы изучили эти элементы в главе 2, то уже можете выполнить 90% заданий, которые лучше всего решаются с помощью регулярных выражений.

Все регулярные выражения, кроме представленных в разделе «Простое», требуют, чтобы был включен режим нечувствительности к регистру символов. В противном случае будут допустимы только символы верхнего регистра. В этом режиме можно будет вместо ‹[A-Za-z]› вводить ‹[A-Z]›, экономя время на нажатиях клавиш.

Как отмечалось в рецепте 2.3, метасимвол ‹\S› является сокращенной формой записи символьного класса. Метасимволу ‹\S› соответствуют любые непробельные символы.

Конструкции ‹@› и ‹\.› соответствуют символу @ и точке соответственно. Поскольку точка является метасимволом, при ее использовании за пределами символьных классов ее необходимо экранировать символом обратного слэша. Символ @ не имеет специального назначения ни в одном из диалектов регулярных выражений, рассматриваемых в этой книге. Перечень всех метасимволов, которые необходимо экранировать, приводится в рецепте 2.1.

‹[A-Z0-9.-]› и другие последовательности, заключенные в квадратные скобки, являются символьными классами. Данному символьному классу соответствует любая буква от A до Z, любая цифра от 0 до 9, а также точка и дефис. Обычно дефис используется в символьных классах для определения диапазона, но когда дефис стоит первым или последним символом в символьном классе, он интерпретируется как литерал. О символьных классах рассказывается в рецепте 2.3, включая возможность комбинирования с сокращенными формами записи, как в этом

случае: ‹[A-Z0-9_!#$%&'*+/=?`{|}~^.-]›. Данному классу соответствует символ слова, а также любой из 19 перечисленных знаков пунктуации.

Когда символы ‹+› и ‹*› используются за пределами символьных классов, они играют роль квантификаторов. Знак плюс повторяет предшествующий ему элемент один или более раз. В этих регулярных выражениях квантифицируемыми элементами являются символьные классы, а иногда и группы. Таким образом, подвыражению ‹[A-Z0-9.-]+› соответствует одна или более букв, цифр, точек и/или дефисов.

Например, группе ‹(?:[A-Z0-9-]+\.)+› соответствуют один или более алфавитных символов, цифр и/или дефисов, за которыми следует один литерал точки. Символ плюс обеспечивает повторение этой группы один или более раз. Группа должна обнаруживать как минимум одно совпадение, но может совпадать столько раз, сколько это возможно. Более подробно механизм работы символа плюс и других квантификаторов описывается в рецепте 2.12.

‹(?:…)› — это несохраняющая группа. Она используется для группировки части регулярного выражения, благодаря чему можно применять квантификатор к группе целиком. Сохраняющая группа ‹(…)› делает то же самое, но имеет более простой синтаксис. Во всех регулярных выражениях, которые использовались до сих пор, можно заменить все комбинации символов ‹(?:› на ‹(›, и результат от этого не изменится. Однако нам не требуется сохранять какие-либо части адресов электронной почты, а несохраняющая группировка обладает большей эффективностью, хотя и делает регулярное выражение менее удобочитаемым. Полная информация о сохраняющих и несохраняющих группах приводится в рецепте 2.9.

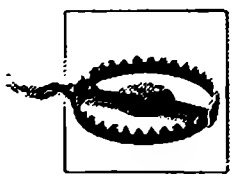
В большинстве диалектов якорные метасимволы ‹^› и ‹$› вынуждают регулярное выражение искать совпадение в начале и в конце испытуемого текста соответственно. Если поместить все регулярное выражение между этими метасимволами, тем самым будет поставлено требование совпадения с регулярным выражением всей испытуемой строки целиком.

Это очень важное условие при проверке ввода пользователя. Едва ли кто-то пожелает принять строку `drop database; -- joe@server.com haha!` в качестве допустимого адреса электронной почты. Без якорных метасимволов эта строка будет соответствовать всем предыдущим регулярным выражениям, поскольку они будут обнаруживать совпадение joe@server.com в середине данной строки. Подробнее об этом рассказывается в рецепте 2.5. Это также объясняет, почему в таких регулярных выражениях должен быть отключен режим «символам ^ и $ соответствуют границы строк».

В диалекте регулярных выражений Ruby символ крышки и знак доллара всегда совпадают с границами строк. Регулярные выражения, использующие символ крышки и знак доллара, будут корректно работать в языке Ruby, но только если проверяемая строка не содержит символов перевода строки. Если строка может содержать символы перевода

строки, все регулярные выражения, использующие метасимволы ‹^› и ‹$›, будут совпадать с адресом электронной почты в строке drop database; -- LFjoe@server.comLF haha!, где обозначение LF представляет символ перевода строки.

Чтобы избежать этого, следует использовать якорные метасимволы ‹\A› и ‹\Z›. Они соответствуют только началу и концу строки независимо от используемых режимов во всех диалектах, рассматриваемых в этой книге, за исключением диалекта JavaScript. Диалект JavaScript вообще не поддерживает метасимволы ‹\A› и ‹\Z›. Описание этих метасимволов можно найти в рецепте 2.5.



Проблема выбора между метасимволами ‹^› и ‹$› и метасимволами ‹\A› и ‹\Z› свойственна для всех регулярных выражений, осуществляющих проверку ввода. Многие из них приводятся в этой книге. Хотя периодически мы будем напоминать об этой проблеме, тем не менее не будем делать это постоянно или приводить отдельные решения для JavaScript и Ruby в каждом рецепте. Во многих случаях мы будем показывать единственное решение, использующее символ крышки и знак доллара, и указывать диалект Ruby в списке совместимых диалектов. Если вы пользуетесь Ruby, не забывайте применять метасимволы ‹\A› и ‹\Z›, если необходимо избежать совпадения с одной строкой в многострочном тексте.

Пошаговое создание регулярного выражения

Этот рецепт демонстрирует, как можно создавать регулярное выражение шаг за шагом. Этот прием особенно удобно применять при использовании интерактивного инструмента тестирования регулярных выражений, такого как RegexBuddy.

Для начала надо загрузить в инструмент набор допустимых и недопустимых примеров данных. В данном случае это могут быть списки допустимых и недопустимых адресов электронной почты.

Затем пишется простое регулярное выражение, которое совпадает со всеми допустимыми адресами. На этом этапе можно игнорировать возможные совпадения с недопустимыми адресами. Конструкция ‹^\S+@\S+$› уже определяет основную структуру адреса электронной почты: локальный элемент адреса, символ @ и доменное имя.

Определив основную структуру текстового шаблона, можно приступать к усовершенствованию каждой части структуры, пока регулярное выражение не прекратит совпадать с недопустимыми данными. Если предполагается применять регулярное выражение к уже существующим данным, это задание можно будет выполнить достаточно быстро. Если предполагается, что регулярное выражение будет использоваться для обработки ввода пользователя, работа над регулярным выражением, пока оно не окажется достаточно строгим, чтобы соответствовать только допустимым данным, может оказаться довольно сложным делом.

Варианты

Если задача состоит не в том, чтобы определять, является ли вся введенная строка адресом электронной почты, а в том, чтобы отыскать адреса в объемном тексте, в регулярном выражении нельзя использовать метасимволы ‹^› и ‹$›. Однако простое удаление их из регулярного выражения не станет правильным решением. В этом случае получившееся регулярное выражение, которое требует, чтобы имя домена верхнего уровня содержало только алфавитные символы, будет совпадать, например, с фрагментом john@doe.com в строке john@doe.com77. Вместо того чтобы привязывать совпадение с регулярным выражением к началу и к концу испытуемого текста, необходимо указать, что начало локального элемента адреса и имя домена верхнего уровня не могут быть частью более длинных слов.

Это легко реализуется с помощью пары метасимволов границ слов. То есть метасимволы ‹^› и ‹$› нужно заменить метасимволом ‹\b›. Например, в результате такой замены выражение ‹^[A-Z0-9+_.-]+@[A-Z0-9.-]+$› превратится в выражение ‹\b[A-Z0-9+_.-]+@[A-Z0-9.-]+\b›.

См. также

Документ RFC 5322 определяет структуру и синтаксис сообщений электронной почты, включая адреса, используемые в сообщениях. Загрузить RFC 5322 можно по адресу *http://www.ietf.org/rfc/rfc5322.txt*.

В Википедии приводится исчерпывающий список имен доменов верхнего уровня по адресу *http://en.wikipedia.org/wiki/List_of_Internet_top-level_domains*.[1]

В главе 8 приводится множество решений для работы со строками URL и адресами Интернета.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.2. Проверка и форматирование телефонных номеров

Задача

Необходимо определить, является ли ввод пользователя номером телефона в одном из общепринятых форматов, используемых в Северной

---

[1] *http://ru.wikipedia.org/wiki/Список_доменов_верхнего_уровня. – Прим. перев.*

Америке, включая код города. В число этих форматов входят: 1234567890, 123-456-7890, 123.456.7890, 123 456 7890, (123) 456 7890 **и их комбинации. Если ввод представляет собой допустимый номер телефона, его следует привести к стандартному формату** (123) 456-7890, **чтобы обеспечить единообразие отображения телефонных номеров.**

Решение

С помощью регулярного выражения можно легко проверить, ввел ли пользователь что-то похожее на допустимый телефонный номер. Применив сохраняющие группы для сохранения каждого набора цифр, то же самое регулярное выражение можно будет использовать для приведения испытуемого текста к желаемому формату.

Регулярное выражение

```
^\(?([0-9]{3})\)?[-.●]?([0-9]{3})[-.●]?([0-9]{4})$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
($1)●$2-$3
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
(\1)●\2-\3
```

Диалекты замещающего текста: Python, Ruby

Пример для C#

```
Regex phoneRegex =
    new Regex(@"^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$");

if (phoneRegex.IsMatch(subjectString)) {
    string formattedPhoneNumber =
    phoneRegex.Replace(subjectString, "($1) $2-$3");
} else {
    // Недопустимый номер телефона
}
```

Пример для JavaScript

```
var phoneRegex = /^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$/;

if (phoneRegex.test(subjectString)) {
    var formattedPhoneNumber =
        subjectString.replace(phoneRegex, "($1) $2-$3");
} else {
    // Недопустимый номер телефона
}
```

Другие языки программирования

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецептах 3.6, где демонстрируется, как проверить совпадение регулярного выражения со всей испытуемой строкой, и 3.15, где демонстрируется, как выполнить замену совпадений с повторным использованием фрагментов совпадений.

Обсуждение

Этому регулярному выражению соответствуют три группы цифр. Первая группа может быть заключена в необязательные круглые скобки, и за первыми двумя группами могут следовать символы-разделители трех видов (дефис, точка или пробел). Ниже приводится структура регулярного выражения, разбитого на отдельные части, с опущенными дополнительными группами цифр:

```
^          # Проверка совпадения с началом строки.
\(         # Соответствует литералу "("
  ?        #   ноль или один раз.
(          # Сохранение вложенного совпадения для обратной ссылки 1:
  [0-9]    #   Соответствует цифре
    {3}    #     точно три раза.
)          # Конец сохраняющей группы 1.
\)         # Соответствует литералу ")"
  ?        #   ноль или один раз.
[-. ]      # Соответствует одному символу: дефису, точке или пробелу
  ?        #   ноль или один раз.
...        # [Соответствует остальным цифрам и разделителю.]
$          # Проверка совпадения с концом строки.
```

Рассмотрим каждую из этих частей более подробно.

Символы ‹^› и ‹$› в начале и в конце регулярного выражения – это метасимволы специального назначения, которые называются *якорными метасимволами*, или *проверками*. Этим метасимволам соответствует не текст, а определенные позиции в тексте. В частности, метасимволу ‹^› соответствует начало текста, а метасимволу ‹$› – конец. Они гарантируют, что регулярное выражение не будет совпадать с более длинным текстом, таким как 123-456-78901.

Как уже неоднократно говорилось, круглые скобки в регулярных выражениях имеют специальное назначение, но в данном случае необходимо разрешить пользователю вводить круглые скобки, и регулярное выражение должно распознавать их. Это наглядный пример ситуации, когда необходимо использовать символ обратного слэша для экранирования специальных символов, чтобы механизм регулярных выражений интерпретировал их как обычные символы. То есть последовательностям ‹\(› и ‹\)›, охватывающим первую группу цифр, соответствуют символы круглых скобок в тексте. Обе последовательности сопровождаются вопросительным знаком, который делает совпадения с ними

необязательными. Подробнее о знаке вопроса мы поговорим после того, как рассмотрим остальные части этого регулярного выражения.

Круглые скобки, перед которыми не стоят символы обратного слэша, образуют сохраняющие группы и используются для запоминания фрагментов текста, совпавших с подвыражениями, заключенными в скобки, чтобы позднее можно было извлечь эти фрагменты. В данном случае обратные ссылки на сохраненные фрагменты используются в замещающем тексте, благодаря чему мы легко можем привести телефонный номер к нужному формату.

В этом регулярном выражении используются еще два типа элементов – символьные классы и квантификаторы. Символьные классы позволяют описать совпадение с любым символом из определенного набора. Конструкция ‹[0-9]› – это символьный класс, которому соответствует одна цифра. Все диалекты регулярных выражений, описываемые в этой книге, позволяют использовать сокращенную форму записи символьного класса ‹\d›, которой также соответствует одна цифра, но в некоторых диалектах метасимволу ‹\d› соответствует цифра из любого набора символов или алфавита, а это не совсем то, что нам в данном случае требуется. Подробнее о метасимволе ‹\d› рассказывается в рецепте 2.3.

Конструкция ‹[-.•]› – это еще один символьный класс, которому соответствует любой из трех допустимых символов-разделителей. Местоположение символа дефиса в символьном классе имеет большое значение. Когда дефис находится между двумя символами, он образует диапазон символов, как в символьном классе ‹[0-9]›. Другой способ обеспечить интерпретацию дефиса в символьном классе как литерала заключается в том, чтобы экранировать его символом обратного слэша. Так, символьный класс ‹[.\-•]› эквивалентен предыдущему. Здесь символ ‹•› представляет пробел.

Наконец, квантификаторы позволяют описать повторяющиеся совпадения с элементом регулярного выражения или группы. ‹{3}› – это квантификатор, который говорит, что предшествующий ему элемент должен совпасть точно три раза. То есть регулярное выражение ‹[0-9]{3}› эквивалентно выражению ‹[0-9][0-9][0-9]›, но оно короче и легче читается. Знак вопроса (упоминавшийся выше) – это квантификатор, который обеспечивает совпадение предшествующего ему элемента ноль или один раз. Этот квантификатор можно также записать как ‹{0,1}›. Любой квантификатор, допускающий нулевое число совпадений предшествующего элемента, фактически делает этот элемент необязательным. Поскольку знак вопроса указан после каждого разделителя, допускается, что цифры в телефонном номере будут следовать непосредственно друг за другом.

Следует заметить, что несмотря на то, что в данном рецепте говорится о форматах представления телефонных номеров, используемых в Северной Америке, фактически он предназначен для работы с номерами в системе NANP (*North American Numbering Plan – северо-*

американская схема присвоения номеров). NANP – это схема присвоения номеров для стран, в которых используется код страны «1». В их число входят Соединенные Штаты и их территории, Канада, Бермудские острова и 17 государств Карибского бассейна. Мексика и государства Центральной Америки не входят в эту систему.

Варианты

Устранение недопустимых номеров

Итак, регулярному выражению соответствуют любые 10-значные номера. Если необходимо ограничиться совпадением только с номерами, соответствующими североамериканской схеме присвоения номеров, необходимо соблюсти следующие требования:

- *Код зоны* должен начинаться с цифры в диапазоне 2–9, за которой следует цифра из диапазона 0–8, а третья цифра может быть любой.
- Вторая группа из трех цифр, известная как *центральная станция*, или *коммутатор*, должна начинаться с цифры в диапазоне 2–9, за которой следуют любые две цифры.
- Последняя группа из четырех цифр, известная как *код станции*, не имеет ограничений.

Эти требования можно легко реализовать с помощью нескольких свободных классов:

```
^\(?([2-9][0-8][0-9])\)?[-.•]?([2-9][0-9]{2})[-.•]?([0-9]{4})$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Помимо только что перечисленных основных правил существуют правила для различных зарезервированных, неприсвоенных и ограниченных номеров. Если не стоит задача отфильтровывать максимально возможное число телефонных номеров, то не надо и пытаться отбрасывать неиспользуемые номера. Новые коды зон, соответствующие вышеперечисленным правилам, появляются регулярно, при этом, даже если телефонный номер соответствует правилам, это еще не значит, что он действительно используется.

Поиск телефонных номеров в документах

Два небольших изменения в предыдущих регулярных выражениях позволят использовать его для поиска телефонных номеров внутри текста большого объема:

```
\(?\b([0-9]{3})\)?[-.•]?([0-9]{3})[-.•]?([0-9]{4})\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Здесь были убраны проверки ‹^› и ‹$›, привязывавшие регулярное выражение к началу и концу текста. На их место были вставлены метасимволы границы слова (‹\b›) для гарантии того, что совпавший фрагмент текста не является частью более длинного числа или слова.

Подобно метасимволам ‹^› и ‹$›, метасимвол ‹\b› также соответствует определенной позиции, а не какому-то фактическому тексту. В частности, метасимволу ‹\b› соответствует позиция между символом слова и либо не символом слова, либо началом и концом текста. Алфавитные, цифровые символы и символ подчеркивания – все они считаются символами слова (рецепт 2.6).

Обратите внимание, что первый метасимвол границы слова стоит после необязательной открывающей скобки. Это важно, так как между двумя не символами слова, например между пробелом и следующей за ним открывающей круглой скобкой, не может быть границы слова. Проверка первой границы слова обретает важное значение, только когда совпавший номер не окружен круглыми скобками, так как позиция между открывающей круглой скобкой и первой цифрой номера всегда будет совпадать с метасимволом границы слова.

Допустимость ведущего символа «1»

Можно предусмотреть возможность наличия в номере необязательной первой цифры «1», обозначающей код страны (который используется всеми странами в регионе, охватываемом североамериканской схемой присвоения номеров), дополнив регулярное выражение, как показано ниже:

```
^(?:\+?1[-.•]?)?\(?([0-9]{3})\)?[-.•]?([0-9]{3})[-.•]?([0-9]{4})$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В дополнение к форматам телефонных номеров, показанных выше, этому регулярному выражению также соответствуют такие строки, как +1 (123) 456-7890 и 1-123-456-7890. Оно использует несохраняющую группу, которая оформляется как ‹(?:···)›. Когда вопросительный знак следует за неэкранированной открывающей круглой скобкой, как в данной конструкции, он не является квантификатором; вместо этого он помогает идентифицировать тип группировки. Стандартные сохраняющие группы требуют от механизма регулярных выражений следить за обратными ссылками, поэтому несохраняющие группы имеют более высокую эффективность, так как совпавший с группой текст не требуется сохранять для обеспечения возможности сослаться на него позднее. Еще одна причина, почему здесь используется несохраняющая группа, состоит в желании сохранить неизменной строку с замещающим текстом. Если бы в регулярное выражение была добавлена сохраняющая группа, пришлось бы в замещающем тексте, показанном выше в этом рецепте, заменить обратную ссылку $1 на $2 (и так далее).

В эту версию регулярного выражения была добавлена конструкция ‹(?:\+?1[-.•]?)?›. Символу «1» в этом шаблоне предшествует необязательный знак плюс, а вслед за символом «1» может следовать один из трех символов-разделителей (дефис, точка или пробел). Вся добавленная несохраняющая группа также объявлена необязательной, но, так как символ «1» внутри группы является обязательным, предшествующий ему символ плюс и следующий за ним символ-разделитель являются недопустимыми, если в номере отсутствует ведущий символ «1».

Допустимость семизначных номеров телефонов

Чтобы позволить совпадение с номерами телефонов, в которых отсутствует код зоны, необходимо первую группу цифр вместе с окружающими их круглыми скобками и следующим за ними символом-разделителем заключить в необязательную несохраняющую группу:

```
^(?:\(?([0-9]{3})\)?[-.•]?)?([0-9]{3})[-.•]?([0-9]{4})$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поскольку теперь код зоны уже не является обязательной частью совпадения, простая замена любого совпадения замещающим текстом «($1)•$2-$3» может, например, дать в результате строку с пустыми круглыми скобками: () 123-4567. Чтобы обойти эту проблему, нужно добавить программный код, который будет проверять наличие совпадения с первой группой и соответствующим образом изменять замещающий текст.

См. также

Рецепт 4.3, в котором демонстрируется, как выполнять проверку телефонных номеров в международном формате.

Как уже говорилось выше, североамериканская схема присвоения номеров (NANP) действует на территории США, Канады, Бермудских островов и 17 государств Карибского бассейна. Дополнительную информацию об этой схеме можно найти на сайте *http://www.nanpa.com*.

Приемы, использовавшиеся в этом рецепте при создании регулярных выражений и текста замены, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.21 демонстрируется, как вставлять текст, совпавший с сохраняющей группой, в текст замены.

4.3. Проверка международных телефонных номеров

Задача

Необходимо проверить корректность международного телефонного номера. Эти номера телефонов должны начинаться со знака плюс, за которым следует код страны, а затем телефонный номер внутри страны.

Решение

Регулярное выражение

```
^\+(?:[0-9]•?){6,14}[0-9]$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пример для JavaScript

```
function validate(phone) {
    var regex = /^\+(?:[0-9] ?){6,14}[0-9]$/;

    if (regex.test(phone)) {
        // Допустимый международный телефонный номер
    } else {
        // Недопустимый международный телефонный номер
    }
}
```

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.6.

Обсуждение

Правила и соглашения, используемые при выводе международных телефонных номеров, значительно варьируются в зависимости от страны, поэтому довольно сложно реализовать надежную проверку международных телефонных номеров без принятия строгого формата. К счастью, существует стандартная форма записи, определяемая документом ITU-T E.123. Данная форма записи требует, чтобы международные номера телефонов включали начальный знак плюс (известный как *символ, соответствующий международному префиксу*), а для отделения групп цифр допускает использовать только пробелы. Несмотря на то что внутри телефонного номера может употребляться символ тильды (~) для обозначения дополнительного сигнала готовности к набору номера, его следует исключить из регулярного выражения, поскольку это всего лишь процедурный элемент (другими словами, он фактически не набирается), и используется он достаточно редко. Согласно системе нумерации телефонных номеров (ITU-T E.164) номер не может состоять

более чем из 15 цифр. Самые короткие международные телефонные номера содержат семь цифр.

Учитывая вышесказанное, рассмотрим регулярное выражение, опять же разбитое на фрагменты. Так как эта версия записана с использованием режима свободного форматирования, литералы символов пробелов были заменены на последовательность ‹\x20›:

```
^          # Проверка совпадения с началом строки.
\+         # Соответствует литералу символа "+".
(?:        # Несохраняющая группа:
  [0-9]    #   Соответствует цифре.
  \x20     #   Соответствует символу пробела
    ?      #     ноль или один раз.
)          # Конец несохраняющей группы.
  {6,14}   #   Повторить предыдущую группу от 6 до 14 раз.
[0-9]      # Соответствует цифре.
$          # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Якорные метасимволы ‹^› и ‹$› по границам регулярного выражения гарантируют, что регулярное выражение будет совпадать со всем испытуемым текстом. Несохраняющей группе, заключенной в конструкцию ‹(?:…)›, соответствует одна цифра, за которой следует необязательный символ пробела. Повторением этой группы с помощью интервального квантификатора ‹{6,14}› гарантируется соблюдение правила о минимальном и максимальном количестве цифр в номере, при этом допускается появление пробела-разделителя в любом месте внутри номера. Второй символьный класс ‹[0-9]› завершает реализацию правила о количестве цифр (увеличивая допустимое число цифр с интервала от 6 до 14 до интервала от 7 до 15) и гарантирует, что телефонный номер не будет заканчиваться пробелом.

Варианты

Проверка международного телефонного номера в формате EPP

```
^\+[0-9]{1,3}\.[0-9]{4,14}(?:x.+)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Это регулярное выражение следует формату записи международных телефонных номеров, определяемому расширяемым протоколом представления информации (Extensible Provisioning Protocol, EPP). Этот протокол был разработан относительно недавно (работа над ним была завершена в 2004 году) и предназначен для обмена информацией между регистратурами и регистраторами доменных имен. Он используется продолжающими расширяться регистратурами доменных имен, включая

.com, *.info*, *.net*, *.org* и *.us*. Особая его значимость обусловлена тем, что формат EPP записи международных телефонных номеров приобретает все большее распространение и признание, благодаря чему он представляет собой серьезный альтернативный формат хранения (и проверки) международных телефонных номеров.

Для представления телефонных номеров протоколом EPP используется формат *+CCC.NNNNNNNNNNxEEEE*, где *C* – это код страны, состоящий из 1–3 цифр, фрагмент *N* может включать до 14 цифр и *E* – это необязательное расширение. Ведущий знак плюс и точка, следующая за кодом страны, являются обязательными элементами. Символ «x» должен присутствовать только при наличии расширения.

См. также

Рецепт 4.2, в котором описываются дополнительные возможности проверки номеров в североамериканской схеме нумерации.

Документ «ITU-T recommendation E.123» («Notation for national and international telephone numbers, e-mail addresses and Web addresses» – форма записи национальных и международных телефонных номеров, адресов электронной почты и веб-адресов) можно загрузить по адресу *http://www.itu.int/rec/T-REC-E.123*. Документ «ITU-T Recommendation E.164» («The international public telecommunication numbering plan» – международная схема нумерации телекоммуникаций) можно загрузить по адресу *http://www.itu.int/rec/T-REC-E.164*. Национальные схемы нумерации можно загрузить по адресу *http://www.itu.int/ITU-T/inr/nnp*.

Документ RFC 5733 определяет синтаксис и семантику контактных идентификаторов EPP, включая международные телефонные номера. Загрузить RFC 5733 можно по адресу *http://tools.ietf.org/html/rfc5733*.

Приемы, использовавшиеся в этом рецепте при создании регулярных выражений и текста замены, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.4. Проверка дат в традиционных форматах

Задача

Необходимо выполнить проверку дат в традиционных форматах mm/dd/yy, mm/dd/yyyy, dd/mm/yy и dd/mm/yyyy. Необходимо использовать простое регулярное выражение, которое просто проверяет, выглядит ли ввод как дата, не пытаясь проверить корректность таких дат, как 31 февраля.

Решение

Решение 1: соответствует любому из указанных форматов представления дат, допуская отсутствие ведущих нулей:

```
^[0-3]?[0-9]/[0-3]?[0-9]/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Решение 2: соответствует любому из указанных форматов представления дат, требуя наличия ведущих нулей:

```
^[0-3][0-9]/[0-3][0-9]/(?:[0-9][0-9])?[0-9][0-9]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Решение 3: соответствует форматам m/d/yy и mm/dd/yyyy. Допускает любые комбинации из одной или двух цифр в номере дня и месяца и две или четыре цифры в номере года:

```
^(1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Решение 4: соответствует формату mm/dd/yyyy, требует наличия ведущих нулей:

```
^(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Решение 5: соответствует форматам d/m/yy и dd/mm/yyyy. Допускает любые комбинации из одной или двух цифр в номере дня и месяца и две или четыре цифры в номере года:

```
^(3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Решение 6: соответствует формату dd/mm/yyyy, требует наличия ведущих нулей:

```
^(3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9])/[0-9]{4}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Решение 7: соответствует любому из этих форматов представления дат с большей точностью, допускает отсутствие ведущих нулей:

```
^(?:(1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])|↵
(3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9]))/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы сделать это регулярное выражение более удобочитаемым, можно переписать его в режиме свободного форматирования:

```
^(?:
  # m/d или mm/dd
  (1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])
|
  # d/m или dd/mm
  (3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9])
)
# /yy или /yyyy
/(?:[0-9]{2})?[0-9]{2}$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Решение 8: соответствует любому из этих форматов представления дат с большей точностью, требует наличия ведущих нулей:

```
^(?:(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])|↵
(3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9]))/[0-9]{4}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Режим свободного форматирования немного упрощает чтение этого выражения:

```
^(?:
  # mm/dd
  (1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])
|
  # dd/mm
  (3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9])
)
# /yyyy
/[0-9]{4}$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Обсуждение

На первый взгляд, проверку чего-то концептуально простого, такого как даты, легко реализовать с помощью регулярного выражения. Но это далеко не так по следующим двум причинам. Во-первых, даты – вещь настолько обыденная, что люди очень небрежно обращаются с их записью. Так, для кого-то запись 4/1 может обозначать первоапрельский «День смеха». Для кого-то другого это может быть первый рабочий день в году[1], если Новый год приходится на пятницу.

[1] 4 января. – *Прим. перев.*

Другая проблема заключается в том, что регулярные выражения не работают с числами напрямую. Невозможно определить в регулярном выражении правило, которое, к примеру, гласило бы: «соответствовать числу в диапазоне от 1 до 31». Регулярные выражения работают с символами. Приходится использовать выражение `‹3[01]|[12][0-9]|0?[1-9]›`, чтобы описать совпадение с цифрой 3, за которой следует цифра 0 или 1, или совпадение с 1 или 2, за которой следует любая цифра, или совпадение с необязательной цифрой 0, за которой следует цифра в диапазоне от 1 до 9. С помощью символьных классов, таких как `‹[1-9]›`, можно определять диапазоны для одной цифры. Это возможно благодаря тому, что символы, обозначающие цифры от 0 до 9, следуют непосредственно друг за другом в таблицах символов ASCII и Юникод. Подробнее о сопоставлении регулярных выражений со всеми типами чисел рассказывается в главе 6.

Вследствие всего вышеизложенного необходимо выбирать между простотой и точностью регулярного выражения. Если заранее известно, что испытуемый текст не содержит ошибочных дат, можно использовать тривиальное регулярное выражение, такое как `‹\d{2}/\d{2}/\d{4}›`. Тот факт, что это выражение совпадает с датами вида 99/99/9999, не имеет большого значения, если такие даты не могут появиться в испытуемом тексте. Такое выражение можно быстро ввести с клавиатуры, и оно отработает очень быстро.

Первые два решения в этом рецепте тоже относятся к простым и быстрым, и они также совпадают с некорректными датами, такими как 0/0/00 и 31/31/2008. В них используются только литералы символов, соответствующие символам-разделителям в датах, и символьные классы (рецепт 2.3), соответствующие цифрам, а также знак вопроса (рецепт 2.12), чтобы обеспечить необязательность некоторых цифр. Регулярное выражение `‹(?:[0-9]{2})?[0-9]{2}›` допускает, что номер года может состоять из двух или четырех цифр. Выражению `‹[0-9]{2}›` соответствуют точно две цифры. Выражению `‹(?:[0-9]{2})?›` соответствуют ноль или две цифры. Несохраняющая группировка (рецепт 2.9) является обязательной, потому что вопросительный знак должен применяться к символьному классу и квантификатору `‹{2}›` одновременно. Выражению `‹[0-9]{2}?›` соответствуют точно две цифры, как и выражению `‹[0-9]{2}›`. Без группировки вопросительный знак делает предшествующий ему квантификатор минимальным, что не оказывает никакого эффекта, потому что квантификатор `‹{2}›` не позволяет повторять предшествующий ему элемент более двух раз или менее двух раз.

В решениях с 3 по 6 номер месяца ограничивается диапазоном от 1 до 12, а число – диапазоном от 1 до 31. Для описания совпадения с разными парами цифр при оформлении диапазонов двузначных чисел применялась конструкция выбора (рецепт 2.8), заключенная в группу. Здесь использовалась сохраняющая группировка, потому что число, месяц и год могут потребоваться позднее.

Последние два решения имеют немного большую сложность, поэтому они представлены в режиме свободного форматирования. Единственное отличие этого режима заключается в большей удобочитаемости. Диалект JavaScript не поддерживает режим свободного форматирования. Заключительные решения поддерживают все форматы представления дат, как и первые два примера. Различие между ними состоит в том, что в последних двух выражениях используется дополнительный уровень конструкции выбора, позволяющий пропускать даты 12/31 и 31/12 и запрещающий недопустимые номера месяцев, такие как 31/31.

Варианты

Если вместо того чтобы определять, является ли введенная строка датой, необходимо отыскать даты в тексте документа, в регулярном выражении нельзя использовать метасимволы ‹^› и ‹$›. Однако простое удаление их из регулярного выражения не является правильным решением. В этом случае получившееся регулярное выражение будет совпадать, например, с фрагментом 12/12/2001 в строке 9912/12/200199. Вместо того чтобы привязывать совпадение с регулярным выражением к началу и к концу испытуемого текста, необходимо указать, что даты не могут быть частью более длинных последовательностей цифр.

Это легко реализуется с помощью пары метасимволов границ слов. В регулярном выражении цифры интерпретируются как символы слов. То есть метасимволы ‹^› и ‹$› нужно заменить метасимволом ‹\b›. Например:

```
\b(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

В этой главе приводится еще несколько рецептов поиска совпадений с датами и временем. В рецепте 4.5 демонстрируется, как более точно проверять даты в традиционных форматах. В рецепте 4.6 рассказывается, как проверять значения времени в традиционных форматах. В рецепте 4.7 показано, как проверять дату и время в формате ISO 8601.

Рецепт 6.7 разъясняет, как создавать регулярные выражения, соответствующие числам в заданном диапазоне.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.5. Точная проверка дат в традиционных форматах

Задача

Необходимо проверить даты в традиционных форматах mm/dd/yy, mm/dd/yyyy, dd/mm/yy и dd/mm/yyyy, как демонстрировалось в рецепте 4.4. Но на этот раз требуется, чтобы регулярное выражение не пропускало недопустимые даты, такие как 31 февраля.

Решение

C#

В первом решении требуется, чтобы сначала шел месяц, потом день. Представленное регулярное выражение совместимо с различными диалектами:

```
^(?<month>[0-3]?[0-9])/(?<day>[0-3]?[0-9])/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10

Далее приводится законченное решение на языке C#:

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
    "^(?<month>[0-3]?[0-9])/(?<day>[0-3]?[0-9])/" +
    "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
    int year = int.Parse(matchResult.Groups["year"].Value);
    if (year < 50) year += 2000;
    else if (year < 100) year += 1900;
    try {
        foundDate = new DateTime(year,
            int.Parse(matchResult.Groups["month"].Value),
            int.Parse(matchResult.Groups["day"].Value));
    } catch {
        // Недопустимая дата
    }
}
```

Во втором решении требуется, чтобы день указывался перед месяцем. Единственное отличие – в этом регулярном выражении переставлены имена сохраняющих групп:

```
^(?<day>[0-3]?[0-9])/(?<month>[0-3]?[0-9])/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10

Программный код на C# не изменился, кроме самого регулярного выражения:

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
    "^(?<day>[0-3]?[0-9])/(?<month>[0-3]?[0-9])/" +
    "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
    int year = int.Parse(matchResult.Groups["year"].Value);
    if (year < 50) year += 2000;
    else if (year < 100) year += 1900;
    try {
        foundDate = new DateTime(year,
            int.Parse(matchResult.Groups["month"].Value),
            int.Parse(matchResult.Groups["day"].Value));
    } catch {
        // Недопустимая дата
    }
}
```

Perl

Первое решение требует, чтобы сначала шел месяц, потом день. Представленное регулярное выражение совместимо со всеми диалектами, рассматриваемыми в этой книге:

```
^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Далее приводится законченное решение на языке Perl:

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
$validdate = 0;
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!)
{
    $month = $1;
    $day = $2;
    $year = $3;
    $year += 2000 if $year < 50;
    $year += 1900 if $year < 100;
    if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 ||
                                          $year % 400 == 0)) {
        $validdate = 1 if $day >= 1 && $day <= 29;
    } elsif ($month >= 1 && $month <= 12) {
        $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];
    }
}
```

Второе решение требует, чтобы день указывался перед месяцем. Регулярное выражение не изменилось. В программном коде на Perl поменялась интерпретация первых двух сохраняющих групп:

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
$validdate = 0;
```

```
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!)
{
    $day = $1;
    $month = $2;
    $year = $3;
    $year += 2000 if $year < 50;
    $year += 1900 if $year < 100;
    if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 ||
                                          $year % 400 == 0)) {
        $validdate = 1 if $day >= 1 && $day <= 29;
    } elsif ($month >= 1 && $month <= 12) {
        $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];
    }
}
```

Проверка исключительно средствами регулярного выражения

Поставленную задачу можно решить исключительно средствами регулярного выражения без дополнительной обработки результатов в программном коде. Данное решение приводится для тех, кто в приложении может изменить только регулярное выражение.

Сначала месяц, потом день:

```
^(?:
  # Февраль (29 дней в любом году)
  (?<month>0?2)/(?<day>[12][0-9]|0?[1-9])
|
  # месяцы продолжительностью 30 дней
  (?<month>0?[469]|11)/(?<day>30|[12][0-9]|0?[1-9])
|
  # месяцы продолжительностью 31 день
  (?<month>0?[13578]|1[02])/(?<day>3[01]|[12][0-9]|0?[1-9])
)
# Год
/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Perl 5.10, Ruby 1.9

```
^(?:
  # Февраль (29 дней в любом году)
  (0?2)/([12][0-9]|0?[1-9])
|
  # месяцы продолжительностью 30 дней
  (0?[469]|11)/(30|[12][0-9]|0?[1-9])
|
  # месяцы продолжительностью 31 день
  (0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9])
)
# Год
/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
^(?:(0?2)/([12][0-9]|0?[1-9])|(0?[469]|11)/(30|[12][0-9]|0?[1-9])|↵
(0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9]))/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Сначала день, потом месяц:

```
^(?:
  # Февраль (29 дней в любом году)
  (?<day>[12][0-9]|0?[1-9])/(?<month>0?2)
|
  # месяцы продолжительностью 30 дней
  (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[469]|11)
|
  # месяцы продолжительностью 31 день
  (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[13578]|1[02])
)
# Год
/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Perl 5.10, Ruby 1.9

```
^(?:
  # Февраль (29 дней в любом году)
  ([12][0-9]|0?[1-9])/(0?2)
|
  # месяцы продолжительностью 30 дней
  (30|[12][0-9]|0?[1-9])/([469]|11)
|
  # месяцы продолжительностью 31 день
  (3[01]|[12][0-9]|0?[1-9])/(0?[13578]|1[02])
)
# Год
/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
^(?:([12][0-9]|0?[1-9])/(0?2)|(30|[12][0-9]|0?[1-9])/([469]|11)|↵
(3[01]|[12][0-9]|0?[1-9])/(0?[13578]|1[02]))/((?:[0-9]{2})?[0-9]{2})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Регулярное выражение с программным кодом

По сути, имеется два способа точной проверки дат с помощью регулярных выражений. Первый метод заключается в использовании простого

регулярного выражения, которое лишь сохраняет группы чисел, выглядящие как комбинация месяц/день/год, с последующей проверкой этих чисел на корректность в программном коде.

Главное преимущество этого метода заключается в простоте наложения дополнительных ограничений, таких как ограничение дат определенным периодом. Во многих языках программирования имеются средства для работы с датами. В решении для языка C# для проверки допустимости даты и преобразования ее в удобный формат за одно действие применяется структура `DateTime`, используемая в платформе .NET.

Мы использовали первое регулярное выражение из рецепта 4.4, которое считает допустимыми любые числа в диапазоне от 0 до 39 для обозначения дня и месяца. Это упрощает переход от формата mm/dd/yyyy к формату dd/mm/yyyy простой заменой сохраняющей группы, которая интерпретируется как номер месяца. При использовании именованных сохранений это означает замену имен сохраняющих групп в регулярном выражении. При использовании нумерованных сохранений производится замена ссылок на нумерованные группы в программном коде.

Проверка исключительно средствами регулярного выражения

Другой метод целиком опирается на использование регулярного выражения. Здесь можно применить тот же прием, основанный на выборе альтернатив, который использовался в заключительных решениях, представленных в рецепте 4.4. Решение становится более или менее простым, если считать каждый год високосным, а дату 29 февраля – допустимой независимо от года. Для ограничения допустимости даты 29 февраля потребовалось бы описать в регулярном выражении, какие годы считаются високосными, а какие нет.

Проблема решения, опирающегося исключительно на использование регулярного выражения, в том, что для большей точности определения оно больше не сохраняет день и месяц в единственной сохраняющей группе. Теперь имеются три сохраняющих группы для месяца и три сохраняющих группы для дня. Когда регулярное выражение обнаруживает совпадение с датой, только три группы из семи будут хранить некоторые данные. Для февраля месяц и день будут сохраняться в группах 1 и 2. Для месяца продолжительностью 30 дней месяц и день будут сохраняться в группах 3 и 4. Для месяца продолжительностью 31 день месяц и день будут сохраняться в группах 5 и 6. Группа 7 всегда будет хранить год.

Только диалекты Perl 5.10, Ruby 1.9 и .NET способны помочь в этой ситуации. В этих диалектах допускается иметь несколько именованных групп с одним и тем же именем. Подробнее об этом рассказывается в рецепте 2.11 в разделе «Группы с одинаковыми именами». Это преимущество дает возможность использовать группы с именами «month» и «day» в каждой из альтернатив. Когда регулярное выражение находит совпа-

дение, остается просто извлечь совпавший текст, используя имена групп «month» и «day», не беспокоясь о том, сколько дней в месяце.

Для всех остальных диалектов используется решение на основе нумерованных сохраняющих групп. Когда обнаруживается совпадение, приходится проверять, в какой из трех групп хранится день, а в какой из трех других групп – месяц.

Решение, опирающееся исключительно на использование регулярного выражения, представляет интерес только в ситуациях, когда регулярное выражение – это единственное, что можно использовать. Например, в приложении, предлагающем только поле для ввода регулярного выражения. При программировании многие вещи гораздо проще решить с помощью дополнительного программного кода. Это особенно удобно, если позднее может потребоваться добавить дополнительные проверки дат.

Варианты

Чтобы показать, насколько сложным может стать регулярное выражение при добавлении дополнительных требований, ниже приводится решение, основанное исключительно на применении регулярного выражения, которому соответствуют даты в диапазоне от 2 мая 2007 года до 29 августа 2008 года в формате d/m/yy или dd/mm/yyyy:

```
# от 2 мая 2007 до 29 августа 2008
^(?:
  # от 2 мая 2007 до 31 декабря 2007
  (?:
    # от 2 мая до 31 мая
    (?<day>3[01]|[12][0-9]|0?[2-9])/(?<month>0?5)/(?<year>2007)
  |
    # от 1 июня до 31 декабря
    (?:
      # месяцы продолжительностью 30 дней
      (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[69]|11)
    |
      # месяцы продолжительностью 31 день
      (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[78]|1[02])
    )
    /(?<year>2007)
  )
|
  # от 1 января 2008 до 29 августа 2008
  (?:
    # от 1 августа до 29 августа
    (?<day>[12][0-9]|0?[1-9])/(?<month>0?8)/(?<year>2008)
  |
    # от 1 января до 30 июня
    (?:
```

```
  # Февраль
  (?<day>[12][0-9]|0?[1-9])/(?<month>0?2)
|
  # месяцы продолжительностью 30 дней
  (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[46])
|
  # месяцы продолжительностью 31 день
  (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[1357])
)
/(?<year>2008)
  )
)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Perl 5.10, Ruby 1.9

См. также

В этой главе приводится еще несколько рецептов поиска совпадений с датами и временем. В рецепте 4.4 демонстрируется, как упростить проверку дат в традиционных форматах за счет некоторой потери точности. В рецепте 4.6 рассказывается, как проверять значения времени в традиционных форматах. В рецепте 4.7 показано, как проверять дату и время в формате ISO 8601.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.6. Проверка времени в традиционных форматах

Задача

Необходимо проверить ввод времени в различных традиционных форматах записи времени, таких как hh:mm и hh:mm:ss в 12-часовом и 24-часовом форматах.

Решение

Часы и минуты в 12-часовом формате:

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9])( ?[AP]M)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часы и минуты в 24-часовом формате:

```
^(2[0-3]|[01]?[0-9]):([0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часы, минуты и секунды в 12-часовом формате:

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9]):([0-5]?[0-9])(\s?[AP]M)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часы, минуты и секунды в 24-часовом формате:

```
^(2[0-3]|[01]?[0-9]):([0-5]?[0-9]):([0-5]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Знаки вопроса во всех предыдущих регулярных выражениях делают ведущие нули необязательными. Чтобы сделать их обязательными, достаточно удалить знаки вопроса.

Обсуждение

Реализовать проверку времени определенно проще, чем проверку дат. Продолжительность каждого часа составляет 60 минут, а продолжительность каждой минуты – 60 секунд. Это означает, что нет необходимости строить сложные конструкции выбора в регулярном выражении. Для минут и секунд конструкция выбора вообще не требуется. Выражению ‹[0-5]?[0-9]› соответствует цифра в диапазоне от 0 до 5, за которой следует цифра в диапазоне от 0 до 9, то есть этому выражению соответствуют числа в диапазоне от 0 до 59. Знак вопроса после первого символьного класса делает его необязательным. Тем самым единственная цифра также рассматривается как допустимое обозначение числа минут или секунд между 0 и 9. Если первые 10 минут или секунд должны записываться как значения от 00 до 09, следует удалить знак вопроса. В рецептах 2.3 и 2.12 подробно описываются символьные классы и квантификаторы, такие как знак вопроса.

Для поиска совпадений со значением часов нам потребуется использовать конструкцию выбора (рецепт 2.8). Вторая цифра считается допустимой в разных диапазонах в зависимости от первой цифры. В 12-часовом формате, если первая цифра 0, вторая цифра может быть любой из 10 цифр, но если первая цифра 1, вторая цифра может быть только 0, 1 или 2. В регулярном выражении соответствующая конструкция записана как ‹1[0-2]|0?[1-9]›. В 24-часовом формате, если первая цифра 0 или 1, вторая цифра может быть любой из 10 цифр, но если первая цифра 2, вторая цифра может быть только цифрой в диапазоне от 0 до 3. В регулярном выражении эта проверка выглядит как ‹2[0-3]|[01]?[0-9]›. Здесь также знак вопроса позволяет записывать первые 10 часов одной цифрой.

Если требуется, чтобы час всегда записывался двумя цифрами, знак вопроса следует удалить.

Мы заключили в круглые скобки части регулярного выражения, соответствующие часам, минутам и секундам, тем самым обеспечив простой способ извлечения цифр, обозначающих часы, минуты и секунды без символов двоеточия. В рецепте 2.9 описывается, как с помощью круглых скобок оформляются сохраняющие группы. В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде.

Круглые скобки, заключающие подвыражение, которому соответствует час, объединяют две альтернативные формы записи. Если удалить эти скобки, регулярное выражение будет работать некорректно. Удаление скобок, окружающих минуты и секунды, не окажет никакого эффекта, кроме того, что сделает невозможным извлекать их значения по отдельности.

В регулярном выражении, осуществляющем поиск значений времени в 12-часовом формате, предусматривается возможность появления индикатора AM или PM. Кроме того, допускается присутствие пробела между значением времени и индикатором AM/PM. Конструкция ‹[AP]M› совпадает с последовательностями символов AM и PM. Подвыражению ‹•?› соответствует необязательный пробел. Группа ‹(•?[AP]M)?› объединяет пробел и индикатор в одно целое и делает их необязательными. Здесь нельзя использовать конструкцию ‹•?([AP]M)?›, потому что она допускает наличие пробела, даже когда индикатор отсутствует.

Варианты

Если вместо того чтобы определять, является ли введенная строка значением времени, необходимо отыскать значения времени в объемном тексте, в регулярном выражении нельзя использовать метасимволы ‹^› и ‹$›. Однако простое удаление их из регулярного выражения не является правильным решением. В этом случае получившееся регулярное выражение будет совпадать, например, с фрагментом 12:12 в строке 9912:1299. Вместо того чтобы привязывать совпадение с регулярным выражением к началу и к концу испытуемого текста, необходимо указать, что значения времени не могут быть частью более длинных последовательностей цифр.

Это легко реализуется с помощью пары метасимволов границ слов. В регулярном выражении цифры интерпретируются как символы слов. То есть метасимволы ‹^› и ‹$› нужно заменить метасимволом ‹\b›. Например:

```
\b(2[0-3]|[01]?[0-9]):([0-5]?[0-9])\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Границы слов не отклоняют все подряд – они отклоняют только символы, цифры и символы подчеркивания. Только что продемонстрированное регулярное выражение, совпадающее с часами и минутами в 24-часовом формате, обнаружит соответствие 16:08 в испытуемом тексте `The time is 16:08:42 sharp`. Пробел не является символом слова, тогда как `1` является им, поэтому граница слова обнаружит совпадение между ними. Цифра `8` является символом слова, а двоеточие – нет, поэтому метасимвол ‹\b› также совпадет с позицией между ними.

Если наряду с символами слова необходимо отклонить и двоеточия, следует использовать проверку соседних символов (рецепт 2.16). Приведенное далее регулярное выражение не совпадет ни с одной частью строки `The time is 16:08:42 sharp`. Оно будет работать только в диалектах, поддерживающих ретроспективную проверку:

```
(?<![:\w])(2[0-3]|[01]?[0-9]):([0-5]?[0-9])(?![:\w])
```

Параметры: нет

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

См. также

В этой главе приводится еще несколько рецептов поиска совпадений с датами и временем. В рецептах 4.4 и 4.5 демонстрируется, как проверять даты в традиционных форматах. В рецепте 4.7 показано, как проверять дату и время в формате ISO 8601.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

4.7. Проверка даты и времени в формате ISO 8601

Задача

Необходимо обеспечить совпадение с датой и/или временем в официальном формате ISO 8601, который является основой для многих стандартизованных форматов представления даты и времени. Например, в языке XML Schema встроенные типы `date`, `time` и `dateTime` основаны на стандарте ISO 8601.

Решение

Стандарт ISO 8601 определяет широкий диапазон форматов представления дат и времени. В большинстве приложений, использующих стан-

дарт ISO 8601, поддерживается только некоторая часть из них. Решения, представленные ниже, реализуют поиск наиболее употребимых форматов, определяемых стандартом ISO 8601. В этот рецепт также были добавлены решения на языке XML Schema, который является частной реализацией стандарта ISO 8601.

Даты

Следующим выражениям соответствует календарный месяц (например, 2008-08). Дефис обязателен:

```
^([0-9]{4})-(1[0-2]|0[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Применение именованных сохранений делает регулярное выражение и программный код, ссылающийся на сохраняющие группы, более удобочитаемыми:

```
^(?<year>[0-9]{4})-(?<month>1[0-2]|0[1-9])$
```

Параметры: нет
Диалекты: .NET, Java 7, XregExp, PCRE 7, Perl 5.10, Ruby 1.9

В диалекте Python используется иной синтаксис определения именованных сохранений – с добавлением символа P. Для краткости мы покажем только одно решение для диалекта Python. Во всех остальных решениях будут использоваться именованные сохранения в стиле .NET, которые легко можно перевести в стиль диалекта Python.

```
^(?P<year>[0-9]{4})-(?P<month>1[0-2]|0[1-9])$
```

Параметры: нет
Диалекты: PCRE, Python

Стандарт ISO 8601 разрешает опускать дефисы в календарных датах, благодаря чему обе строки, 2008-08-20 и 20080820, представляют одну и ту же дату. Следующее регулярное выражение учитывает это, а также считает допустимыми формы записи YYYY-MMDD и YYYYMM-DD, которые не соответствуют стандарту ISO 8601:

```
^([0-9]{4})-?(1[0-2]|0[1-9])-?(3[01]|0[1-9]|[12][0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?(?<month>1[0-2]|0[1-9])-?↵
(?<day>3[01]|0[1-9]|[12][0-9])$
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Календарная дата, например 2008-08-30 или 20080830. Дефис необязателен. Это регулярное выражение использует сохраняющую группу и об-

ратную ссылку для поиска форм записи YYYY-MM-DD и YYYYMMDD, но исключая формы записи YYYY-MMDD и YYYYMM-DD.

```
^([0-9]{4})(-?)(1[0-2]|0[1-9])\2(3[01]|0[1-9]|[12][0-9])$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})(?<hyphen>-?)(?<month>1[0-2]|0[1-9])↵
\k<hyphen>(?<day>3[01]|0[1-9]|[12][0-9])$
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Синтаксис именованных обратных ссылок в диалекте Python также отличается:

```
^(?P<year>[0-9]{4})(?P<hyphen>-?)(?P<month>1[0-2]|0[1-9])↵
(?P=hyphen)(?<day>3[01]|0[1-9]|[12][0-9])$
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Дата с номером дня в году (например, 2008-243). Дефис необязателен:

```
^([0-9]{4})-?(36[0-6]|3[0-5][0-9]|[12][0-9]{2}|0[1-9][0-9]|00[1-9])$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?↵
(?<day>36[0-6]|3[0-5][0-9]|[12][0-9]{2}|0[1-9][0-9]|00[1-9])$
```

Параметры: нет

Диалекты: .NET, Java 7, PCRE 7, Perl 5.10, Ruby 1.9

Недели

Неделя года (например, 2008-W35). Дефис необязателен:

```
^([0-9]{4})-?W(5[0-3]|[1-4][0-9]|0[1-9])$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|[1-4][0-9]|0[1-9])$
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Неделя и день в неделе (например, 2008-W35-6). Дефис необязателен:

```
^([0-9]{4})-?W(5[0-3]|[1-4][0-9]|0[1-9])-?([1-7])$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|[1-4][0-9]|0[1-9])-?(?<day>[1-7])$
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Время

Часы и минуты (например, 17:21). Двоеточие необязательно:

```
^(2[0-3]|[01][0-9]):?([0-5][0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01][0-9]):?(?<minute>[0-5][0-9])$
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Часы, минуты и секунды (например, 17:21:59). Двоеточие необязательно:

```
^(2[0-3]|[01][0-9]):?([0-5][0-9]):?([0-5][0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01][0-9]):?(?<minute>[0-5][0-9]):?↵
(?<second>[0-5][0-9])$
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Указатель часового пояса (например, Z, +07 или +07:00). Двоеточие и минуты необязательны:

```
^(Z|[+-](?:2[0-3]|[01][0-9])(?::?(?:[0-5][0-9]))?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часы, минуты и секунды с указателем часового пояса (например, 17:21:59+07:00). Все двоеточия являются необязательными. Минуты в указателе часового пояса также являются необязательными:

```
^(2[0-3]|[01][0-9]):?([0-5][0-9]):?([0-5][0-9])↵
(Z|[+-](?:2[0-3]|[01][0-9])(?::?(?:[0-5][0-9]))?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01][0-9]):?(?<minute>[0-5][0-9]):?(?<second>[0-5][0-9])↵
(?<timezone>Z|[+-](?:2[0-3]|[01][0-9])(?::?(?:[0-5][0-9]))?)$
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Дата и время

Даты с часами, минутами и секундами (например, 2008-08-30 17:21:59 или 20080830 172159). Пробел между датой и временем обязателен. Дефисы и двоеточия необязательны. Следующее регулярное выражение находит значения даты и времени, в которых какие-то дефисы и двоеточия могут быть опущены, вопреки требованиям стандарта ISO 8601.

```
^([0-9]{4})-?(1[0-2]|0[1-9])-?(3[01]|0[1-9]|[12][0-9])↵
*(2[0-3]|[01][0-9]):?([0-5][0-9]):?([0-5][0-9])$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?(?<month>1[0-2]|0[1-9])-?↵
(?<day>3[01]|0[1-9]|[12][0-9])*(?<hour>2[0-3]|[01][0-9])↵
:?(?<minute>[0-5][0-9]):?(?<second>[0-5][0-9])$
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Решение усложняется, если потребовать, чтобы регулярное выражение совпадало только со значениями даты и времени, в которых дефисы и двоеточия либо все присутствуют, либо все отсутствуют. Наиболее очевидное решение достигается за счет использования условных конструкций. Но условные конструкции поддерживаются не всеми диалектами.

```
^([0-9]{4})(-)?(1[0-2]|0[1-9])(?(2)-)(3[01]|0[1-9]|[12][0-9])↵
*(2[0-3]|[01][0-9])(?(2):)([0-5][0-9])(?(2):)([0-5][0-9])$
```

Параметры: нет

Диалекты: .NET, PCRE, Perl, Python

```
^(?<year>[0-9]{4})(?<hyphen>-)?(?<month>1[0-2]|0[1-9])↵
(?(hyphen)-)(?<day>3[01]|0[1-9]|[12][0-9])*(?<hour>2[0-3]|[01][0-9])↵
(?(hyphen):)(?<minute>[0-5][0-9])(?(hyphen):)(?<second>[0-5][0-9])$
```

Параметры: нет

Диалекты: .NET, PCRE 7, Perl 5.10

```
^(?P<year>[0-9]{4})(?P<hyphen>-)?(?P<month>1[0-2]|0[1-9])↵
(?(hyphen)-)(?P<day>3[01]|0[1-9]|[12][0-9])*(?P<hour>2[0-3]|[01][0-9])↵
(?(hyphen):)(?P<minute>[0-5][0-9])(?(hyphen):)(?P<second>[0-5][0-9])$
```

Параметры: нет

Диалекты: PCRE, Perl 5.10, Python

Если используемый вами диалект не поддерживает условный оператор, можно применить оператор выбора для описания альтернатив с разделителями и без них.

```
^(?:([0-9]{4})-?(1[0-2]|0[1-9])-?(3[01]|0[1-9]|[12][0-9])↵
*(2[0-3]|[01][0-9]):?([0-5][0-9]):?([0-5][0-9])|↵
([0-9]{4})(1[0-2]|0[1-9])(3[01]|0[1-9]|[12][0-9])↵
*(2[0-3]|[01][0-9])([0-5][0-9])([0-5][0-9]))$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Даты и время на языке XML Schema

Типы значений для представления дат и времени в стандарте XML Schema основаны на стандарте ISO 8601. В этом языке допускается указывать отрицательный год в датах, предшествовавших началу календаря (до нашей эры). Также допускается указывать годы, состоящие из

четырех и более цифр, но не меньше. Годы, состоящие из более чем четырех цифр, должны начинаться с нуля. Если требуется обеспечить совпадение только с четырехзначными годами, как в предыдущих решениях, удалите `-?(?:[1-9][0-9]*)?` из решений, следующих ниже.

Дата с необязательным указателем часового пояса (например, `2008-08-30` или `2008-08-30+07:00`). Дефисы являются обязательными. Это тип данных `date` в языке XML Schema:

```
^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])↵
(Z|[+-](?:2[0-3]|[01][0-9]):[0-5][0-9])?$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>-?(?:[1-9][0-9]*)?[0-9]{4})-(?<month>1[0-2]|0[1-9])-↵
(?<day>3[01]|0[1-9]|[12][0-9])↵
(?<timezone>Z|[+-](?:2[0-3]|[01][0-9]):[0-5][0-9])?$
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Время с необязательными долями секунды и часовым поясом (например, `01:45:36` или `01:45:36.123+07:00`). Нет ограничения на число цифр для долей секунды. Это тип данных `time` в языке XML Schema:

```
^(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?↵
(Z|[+-](?:2[0-3]|[01][0-9]):[0-5][0-9])?$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01][0-9]):(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])↵
(?<frac>\.[0-9]+)?(?<timezone>Z|[+-](?:2[0-3]|[01][0-9]):[0-5][0-9])?$
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Дата и время с необязательными долями секунды и часовым поясом (например, `2008-08-30T01:45:36` или `2008-08-30T01:45:36.123Z`). Это тип данных `dateTime` в языке XML Schema:

```
^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])↵
T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?↵
(Z|[+-](?:2[0-3]|[01][0-9]):[0-5][0-9])?$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>-?(?:[1-9][0-9]*)?[0-9]{4})-(?<month>1[0-2]|0[1-9])-↵
(?<day>3[01]|0[1-9]|[12][0-9])T(?<hour>2[0-3]|[01][0-9]):↵
(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])(?<ms>\.[0-9]+)?↵
(?<timezone>Z|[+-](?:2[0-3]|[01][0-9]):[0-5][0-9])?$
```

Параметры: нет

Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Обсуждение

Стандарт ISO 8601 определяет широкий диапазон форматов представления даты и времени. Регулярные выражения, представленные здесь, охватывают наиболее часто используемые форматы, но в большинстве систем, использующих стандарт ISO 8601, применяют только ограниченный набор. Например, в языке XML Schema дефисы и двоеточия в представлениях даты и времени являются обязательными элементами. Чтобы обеспечить обязательность дефисов и двоеточий, достаточно просто удалить знак вопроса, стоящий после них. Чтобы запретить использование дефисов и двоеточий, нужно удалить эти символы и знаки вопроса, следующие за ними. Будьте внимательны с несохраняющими группами, которые определяются как ‹(?:···)›. Если знак вопроса и символ двоеточия следуют за открывающей круглой скобкой, эти три символа образуют открывающую скобку несохраняющей группы.

Все числовые части регулярного выражения мы заключили в круглые скобки, упростив тем самым возможность извлечения года, месяца, дня, часов, минут, секунд и часового пояса. В рецепте 2.9 описывается, как с помощью круглых скобок создаются сохраняющие группы. В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с этими сохраняющими группами, в программном коде.

Кроме того, для большинства регулярных выражений демонстрируется альтернатива, основанная на применении именованного сохранения. Некоторые из приведенных форматов представления даты и времени могут оказаться незнакомыми для вас или ваших коллег. Именованное сохранение делает регулярные выражения проще для понимания. Диалекты .NET, Java 7, XRegExp, PCRE 7, Perl 5.10 и Ruby 1.9 поддерживают синтаксис ‹(?<*name*>···)›. Все версии PCRE и Python, рассматриваемые в этой книге, поддерживают альтернативный синтаксис ‹(?P<*name*>···)›, где присутствует дополнительный символ ‹P›. Подробнее об этом рассказывается в рецептах 2.11 и 3.9.

Во всех регулярных выражениях введены ограничения на числовые диапазоны. Например, дни месяца ограничены диапазоном чисел от 01 до 31. Выражения никогда не совпадут с 32-м числом или 13-м месяцем. Ни одно из приведенных регулярных выражений не пытается исключить недопустимые комбинации числа и месяца, такие как 31 февраля. В рецепте 4.5 описывается, как решить эту проблему.

Отдельные дефисы и двоеточия, которые не имеют точного соответствия стандарту ISO 8601, в регулярных выражениях объявлены необязательными, кроме выражений, что приводятся в разделе решений для языка XML Scheme. Например, фрагмент 1733:26 согласно стандарту ISO 8601 не является допустимой формой записи времени, но он будет принят регулярными выражениями. Требование, чтобы все дефисы и двоеточия либо присутствовали, либо отсутствовали, немного усложняет регулярное выражение.

Если все разделители будут одинаковыми, мы можем упростить регулярное выражение, используя сохраняющую группу для первого разделителя и обратные ссылки на остальные разделители. Пример такого подхода приводится выше в подразделе «Даты» в разделе «Решение». Для поиска первого дефиса применяется конструкция ‹(-?)›, ‹(?<hyphen>-?)› или ‹(?P<hyphen>-?)›, соответствующая необязательному дефису и сохраняющая его в именованной или нумерованной группе. Если дефис отсутствует, группа сохранит строку нулевой длины. Знак вопроса, делающий дефис необязательным, должен находиться внутри группы. Если сделать необязательной всю группу, обратные ссылки на эту группу будут терпеть неудачу, если сопоставление с дефисом не будет найдено, так как группа вообще не будет участвовать в сопоставлении. Для поиска совпадений с оставшимися дефисами в решениях применялась конструкция ‹\2›, ‹\k<hyphen>› или ‹(?P=hyphen)›, совпадающая с тем же текстом, с которым совпала сохраняющая группа – либо дефис, либо пустая строка в зависимости от наличия первого дефиса. Применяя нумерованное сохранение, необходимо убедиться в правильности номера для обратной ссылки.

Если все разделители будут разными, как, например, в решении, выполняющем поиск дат и времени одновременно, регулярное выражение становится более сложным. Пример такого подхода приводится выше в подразделе «Даты и время» в разделе «Решение». На этот раз для сопоставления с дефисом используется конструкция ‹(-)?›, ‹(?<hyphen>-)?› или ‹(?P<hyphen>-)?›. Теперь знак вопроса находится за пределами сохраняющей группы, чтобы она вообще не участвовала в сопоставлении при отсутствии дефиса. Это позволяет использовать сохраняющую группу с условием. Конструкция ‹(?(2)-)› соответствует дефису, а конструкция ‹(?(2):)› соответствует двоеточию, если вторая сохраняющая группа участвовала в сопоставлении. Условные операторы в данном случае не имеют альтернатив. Это означает, что они будут совпадать ни с чем (причем это совпадение будет считаться успешным), если вторая сохраняющая группа не участвовала в сопоставлении. Конструкции ‹(?(hyphen)-)› и ‹(?(hyphen):)› делают то же самое, но с использованием именованного сохранения.

Немногие диалекты поддерживают условные операторы. Если диалект не поддерживает условные конструкции, единственное решение заключается в использовании оператора выбора с двумя вариантами – с разделителем и без него. Недостаток такого решения заключается в наличии двух сохраняющих групп для каждого элемента данных – даты и времени. Только одно из двух множеств сохраняющих групп будет участвовать в сопоставлении. Программный код, использующий такое регулярное выражение, будет вынужден проверить обе группы.

См. также

В этой главе приводится еще несколько рецептов поиска совпадений с датами и временем. В рецептах 4.4 и 4.5 демонстрируется, как проверять даты в традиционных форматах. В рецепте 4.6 рассказывается, как проверять значения времени в традиционных форматах.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.11 описываются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.17 демонстрируется применение условных конструкций.

4.8. Ограничение возможности ввода алфавитно-цифровыми символами

Задача

В приложении требуется ограничить пользователя возможностью ввода только алфавитно-цифровые символы латинского алфавита (буквы A–Z и a–z и цифры 0–9).

Решение

Благодаря регулярным выражениям решение получается тривиально простым. Сначала создается символьный класс, включающий требуемый диапазон символов. Затем добавляются квантификатор, повторяющий символьный класс один или более раз, и якорные метасимволы, привязывающие совпадение к началу и к концу строки. Вот и все.

Регулярное выражение

```
^[A-Z0-9]+$
```

Параметры: нечувствительность к регистру символов

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пример для Ruby

```
if subject =~ /^[A-Z0-9]+$/i
    puts "Текст содержит только алфавитно-цифровые символы"
else
    puts "Текст содержит не только алфавитно-цифровые символы"
end
```

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.6. В рецепте 3.4 демонстрируется, как устанавливать параметры регулярных выражений, включая модификатор «нечувствительности к регистру символов».

Обсуждение

Рассмотрим это регулярное выражение, разделив его на четыре части:

```
^          # Проверка совпадения с началом строки.
[A-Z0-9] # Соответствует символу от "A" до "Z" или от "0" до "9"...
+          # от одного до неограниченного числа раз.
$          # Проверка совпадения с концом строки.
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Проверки ‹^› и ‹$› в начале и в конце регулярного выражения гарантируют, что будет проверена вся введенная строка. Без них регулярное выражение могло бы совпасть лишь с частью более длинной строки, пропустив недопустимые символы. Квантификатор ‹+› повторяет предшествующий ему элемент один или более раз. Если потребуется обеспечить совпадение регулярного выражения с пустой строкой, можно просто заменить квантификатор ‹+› на ‹*›. Квантификатор ‹*› выполняет ноль или более повторений, делая предшествующий ему элемент необязательным.

Варианты

Ограничение возможности ввода символами ASCII

Следующее регулярное выражение ограничивает возможность ввода 128 символами из 7-битовой таблицы ASCII. В их число входят 33 неотображаемых символа:

```
^[\x00-\x7F]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ограничение возможности ввода неуправляющими символами ASCII и символами разрыва строки

Следующее регулярное выражение можно использовать, чтобы обеспечить возможность ввода только отображаемых и пробельных символов из таблицы ASCII, исключая управляющие символы. Символы перевода строки и возврата каретки (с кодами 0x0A и 0x0D соответственно) являются наиболее часто используемыми управляющими символами, поэтому они явно были добавлены с помощью метасимволов ‹\n› (перевод строки) и ‹\r› (возврат каретки):

```
^[\n\r\x20-\x7E]+$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ограничение возможности ввода символами, общими для кодировок ISO-8859-1 и Windows-1252

ISO-8859-1 и Windows-1252 (часто называется ANSI) – это две наиболее широко используемые кодировки восьмибитовых символов, основанные на стандарте Latin-1 (если быть более точными, ISO/IEC 8859-1). Однако они несовместимы между собой по символам с кодами от 0x80 до 0x9F. В кодировке ISO-8859-1 эти коды отведены под управляющие символы, тогда как в кодировке Windows-1252 они используются для определения расширенного диапазона символов и знаков пунктуации. Эти отличия порой приводят к сложностям в отображении символов, особенно в документах, не объявляющих свою кодировку, или когда получатель текста использует операционную систему, отличную от Windows. Следующее регулярное выражение может использоваться для ограничения ввода набором символов, общих для кодировок ISO-8859-1 и Windows-1252 (включая общие управляющие символы):

```
^[\x00-\x7F\xA0-\xFF]+$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричная форма записи может осложнить чтение этого регулярного выражения, но данный символьный класс действует точно так же, как и символьный класс ‹[A-Z0-9]›, показанный выше. Ему соответствуют символы в двух диапазонах: \x00-\x7F и \xA0-\xFF.

Ограничение ввода алфавитно-цифровыми символами из любого языка

Это регулярное выражение ограничивает возможность ввода алфавитно-цифровыми символами из любого языка или алфавита:

```
^[\p{L}\p{M}\p{Nd}]+$
```

Параметры: нет

Диалекты: .NET, Java, XRegExp, PCRE, Perl, Ruby 1.9

В нем используется символьный класс, который включает свойства для всех кодовых пунктов Юникода, относящихся к категориям букв (Letter), комбинационных знаков (Mark) или десятичных цифр (Decimal Number), что в точности соответствует определению алфавитно-цифрового символа в стандарте Юникода. Категория комбинационных знаков (Mark) включена в регулярное выражение, так как такие знаки используются во многих языках. Комбинационные знаки – это кодовые пункты, предназначенные для объединения с другими символами (например, надстрочный знак над основным символом).

К сожалению, категории Юникода поддерживаются не всеми диалектами регулярных выражений, рассматриваемыми в этой книге. В частности, это регулярное выражение не будет работать в JavaScript (без использования библиотеки XRegExp), Python и Ruby 1.8. Кроме того, чтобы это регулярное выражение работало в диалекте PCRE, необходимо, чтобы библиотека PCRE была скомпилирована с поддержкой UTF-8. Категории Юникода могут использоваться в семействе функций preg языка PHP (которые опираются на библиотеку PCRE) при добавлении флага /u в конец регулярного выражения.

Следующее регулярное выражение демонстрирует обходное решение для диалекта Python:

```
^[^\W_]+$
```

Параметры: поддержка Юникода
Диалект: Python

Отсутствие поддержки категорий Юникода в диалекте Python компенсируется использованием флага UNICODE или U при создании регулярного выражения. Этот флаг изменяет действие некоторых метасимволов регулярных выражений, вынуждая их использовать таблицу символов Юникода. Метасимвол ‹\w› составляет основную часть решения, поскольку ему соответствуют алфавитно-цифровые символы и символ подчеркивания. Используя его инвертированную версию ‹\W› в инвертированном символьном классе, можно удалить символ подчеркивания из допустимого набора символов. Двойное отрицание, как в этом выражении, иногда бывает весьма полезно в регулярных выражениях, хотя и осложняет их понимание.[1] В Python 3.x поддержка не-ASCII символов включена в метасимволы, такие как ‹\w›, по умолчанию, благодаря чему не требуется указывать флаг UNICODE.

См. также

В рецепте 4.9 демонстрируется, как ограничить уже не набор символов, а длину текста.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 демонстрируется, как организовать сопоставление с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.7 рассказывается о сопоставлении с символами Юникода. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

---

[1] Для пущей забавы (если это попадает под ваше понятие забавы) попробуйте создать три, четыре или даже больше уровней отрицания, добавив негативную проверку соседних символов (рецепт 2.16) и разность символьных классов (раздел «Особенности, характерные для разных диалектов» в рецепте 2.3).

4.9. Ограничение длины текста

Задача

Необходимо убедиться, что строка имеет длину от 1 до 10 символов и содержит только символы от A до Z.

Решение

Все языки программирования, рассматриваемые в этой книге, предоставляют простой и эффективный способ проверки длины текста. Например, строки в JavaScript имеют свойство `length`, которое хранит целое число, определяющее длину строки. Однако в некоторых ситуациях бывает удобно проверять длину строки в регулярных выражениях, особенно когда длина является одним из нескольких факторов, определяющих совпадение испытуемого текста с требуемым шаблоном. Следующее регулярное выражение гарантирует совпадение со строкой, длина которой составляет от 1 до 10 символов, и эта строка содержит только заглавные буквы от A до Z. Это регулярное выражение можно модифицировать под собственные нужды, установив иные значения минимальной и максимальной длины, или сделать допустимыми символы из других диапазонов, отличных от A–Z.

Регулярное выражение

```
^[A-Z]{1,10}$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пример для Perl

```
if ($ARGV[0] =~ /^[A-Z]{1,10}$/) {
    print "Допустимый ввод\n";
} else {
    print "Недопустимый ввод\n";
}
```

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.6.

Обсуждение

Ниже это очень простое регулярное выражение разложено на составляющие:

```
^         # Проверка совпадения с началом строки.
[A-Z]     # Соответствует символу в диапазоне от "A" до "Z"...
  {1,10}  # от 1 до 10 раз.
$         # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Якорные метасимволы ‹^› и ‹$› гарантируют, что будет проверена вся испытуемая строка, в противном случае регулярное выражение могло бы совпасть лишь с 10 символами более длинной строки. Символьному классу ‹[A-Z]› соответствует любой символ верхнего регистра в диапазоне от A до Z, а интервальный квантификатор ‹{1,10}› повторяет символьный класс от 1 до 10 раз. Благодаря комбинации интервального квантификатора с якорными метасимволами, совпадающими с началом и концом строки, регулярное выражение будет терпеть неудачу, если длина испытуемого текста выйдет за указанные пределы.

Примечательно, что символьный класс ‹[A-Z]› явно допускает наличие в строке только заглавных букв. Если в число допустимых потребуется добавить строчные буквы в диапазоне от a до z, достаточно будет просто заменить этот символьный класс на ‹[A-Za-z]› или применить режим нечувствительности к регистру символов. В рецепте 3.4 демонстрируется, как это сделать.

Многие, начиная осваивать регулярные выражения, часто допускают ошибку, пытаясь сэкономить на вводе с клавиатуры, определяя символьный класс ‹[A-z]›. На первый взгляд, такая конструкция выглядит как удачный трюк, позволяющий определить диапазон всех символов верхнего и нижнего регистров. Однако между диапазонами A–Z и a–z в таблице символов ASCII имеется несколько символов пунктуации. Поэтому символьный класс ‹[A-z]› в действительности является эквивалентом символьного класса ‹[A-Z[\]^_`a-z]›.

Варианты

Ограничение длины для произвольного шаблона

Поскольку такие квантификаторы, как ‹{1,10}›, применяются только к предшествующему элементу, ограничение количества символов, соответствующих шаблону, который включает более одного элемента, требует иного подхода.

Как описывалось в рецепте 2.16, опережающие проверки (и противоположные им ретроспективные проверки) являются проверками специального вида, которые, как и метасимволы ‹^› и ‹$›, совпадают с позициями внутри испытуемой строки и не поглощают никакие символы. Опережающие проверки могут быть позитивными и негативными, то есть они могут проверять наличие или отсутствие совпадения с шаблоном за текущей позицией. Позитивная опережающая проверка, записываемая как ‹(?=···)›, может использоваться в начале шаблона, чтобы проверить, попадает ли длина строки в заданный диапазон. Остальная часть регулярного выражения может проверять соответствие требуемо-

му шаблону, не беспокоясь о длине текста. Ниже приводится простой пример реализации такого приема:

```
^(?=.{1,10}$).*
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
^(?=[\S\s]{1,10}$)[\S\s]*
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Очень важно, что якорный метасимвол ‹$› находится внутри опережающей проверки, потому что проверка максимальной длины будет работать, только если гарантируется отсутствие дополнительных символов сверх указанного предела. Так как опережающая проверка в начале регулярного выражения уже ограничивает длину испытуемого текста, следующий за ней шаблон может накладывать другие ограничения. В данном случае шаблон ‹.*› (или ‹[\S\s]*› в версии, добавляющей поддержку JavaScript) используется, чтобы просто обеспечить совпадение с испытуемым текстом без дополнительных ограничений.

Первое регулярное выражение должно использоваться в режиме «точке соответствуют границы строк», чтобы обеспечить корректную работу регулярного выражения, когда испытуемая строка содержит разрывы строк. Как активировать этот режим в разных языках программирования, подробно рассказывается в рецепте 3.4. В JavaScript (без использования библиотеки XRegExp) отсутствует поддержка режима «точке соответствуют границы строк», поэтому во втором регулярном выражении используется символьный класс, совпадающий с любым символом. Дополнительная информация приводится в разделе «Любой символ, включая символы конца строки» в рецепте 2.4.

Ограничение числа непробельных символов

Следующему регулярному выражению соответствует любая строка, содержащая от 10 до 100 непробельных символов:

```
^\s*(?:\S\s*){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

По умолчанию в диалектах NET, JavaScript, Perl и Python 3.x метасимвол ‹\s› соответствует всем пробельным символам Юникода, а метасимвол ‹\S› – всем остальным символам. В диалектах Java, PCRE, Python 2.x и Ruby метасимволу ‹\s› соответствуют только пробельные символы ASCII, а метасимволу ‹\S› – все остальные символы. В Python 2.x можно обеспечить совпадение метасимвола ‹\s› со всеми пробельными символами Юникода, передав флаг `UNICODE` или `U` при создании регулярного выражения. В Java 7 того же эффекта можно добиться, передав флаг `UNICODE_CHARACTER_CLASS`. В диалектах Java (версии с 4 по 6), PCRE

и Ruby 1.9, если необходимо исключить учет пробельных символов Юникода при определении предельного числа символов, можно применить следующую версию регулярного выражения, в которой используются свойства Юникода (описываются в рецепте 2.7):

```
^[\p{Z}\s]*(?:[^\p{Z}\s][\p{Z}\s]*){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Ruby 1.9

Чтобы это регулярное выражение работало в диалекте PCRE, библиотека PCRE должна быть скомпилирована с поддержкой UTF-8. В языке PHP включить поддержку UTF-8 можно с помощью модификатора шаблона /u.

Это последнее регулярное выражение объединяет свойство Separator Юникода, ‹\p{Z}›, с метасимволом ‹\s›, совпадающим с любым пробельным символом. Диапазоны символов, соответствующих ‹\p{Z}› и ‹\s›, не перекрываются полностью. В диапазон ‹\s› входят символы с кодами от 0x09 до 0x0D (табуляция, перевод строки, вертикальная табуляция, перевод формата и возврат каретки), которые согласно стандарту Юникода не имеют свойства Separator. Объединение метасимволов ‹\p{Z}› и ‹\s› в символьный класс гарантирует совпадение с любыми пробельными символами.

В обоих регулярных выражениях квантификатор ‹{10,100}› применяется к предшествующей ему несохраняющей группе, а не к единственному элементу. Группе соответствует любой непробельный символ, за которым могут следовать ноль или более пробельных символов. Интервальный квантификатор способен надежно проследить количество совпавших непробельных символов, потому что в каждой итерации возможно совпадение только с одним непробельным символом.

Ограничение числа слов

Следующее регулярное выражение очень похоже на предыдущий пример ограничения числа непробельных символов, за исключением того, что здесь выполняется подсчет слов, а не отдельных непробельных символов. Оно совпадает с текстом, содержащим от 10 до 100 слов, пропуская любые символы, не являющиеся символами слова, включая знаки пунктуации и пробельные символы:

```
^\W*(?:\w+\b\W*){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В диалектах Java (версии с 4 по 6), JavaScript, PCRE, Python 2.x и Ruby метасимвол ‹\w›, обозначающий символ слова, в этом регулярном выражении будет совпадать только с символами A–Z, a–z, 0–9 и символом подчеркивания _. Поэтому он не может использоваться для подсчета слов, содержащих буквы и цифры, не входящие в набор ASCII. В диа-

лектах .NET и Perl метасимвол ‹\w› опирается на таблицу символов Юникода (так же как и его инвертированная версия ‹\W›, и граница слова ‹\b›) и будет совпадать со всеми буквами и цифрами из всех алфавитов Юникода. В диалекте Python 2.x имеется возможность выбирать, будут эти метасимволы использовать Юникод или нет в зависимости от того, устанавливается ли флаг UNICODE или U при создании регулярного выражения. В Python 3.x они используют Юникод по умолчанию. В Java 7 имеется возможность выбирать, будут эти метасимволы использовать Юникод или нет, с помощью флага UNICODE_CHARACTER_CLASS. Метасимвол ‹\b› в Java всегда использует Юникод.

Если необходимо подсчитать количество слов, содержащих буквы и цифры, не входящие в набор ASCII, эту возможность обеспечат следующие регулярные выражения для дополнительных диалектов регулярных выражений.

```
^[^\p{L}\p{M}\p{Nd}\p{Pc}]*(?:[\p{L}\p{M}\p{Nd}\p{Pc}]+↵
\b[^\p{L}\p{M}\p{Nd}\p{Pc}]*){10,100}$
```

Параметры: нет

Диалекты: .NET, Java, Perl

```
^[^\p{L}\p{M}\p{Nd}\p{Pc}]*(?:[\p{L}\p{M}\p{Nd}\p{Pc}]+↵
(?:[^\p{L}\p{M}\p{Nd}\p{Pc}]+|$)){10,100}$
```

Параметры: нет

Диалекты: .NET, Java, XRegExp, PCRE, Perl, Ruby 1.9

Для выполнения этой работы библиотека PCRE должна быть скомпилирована с поддержкой UTF-8. Включить поддержку UTF-8 в языке PHP можно с помощью модификатора шаблона /u.

Как было отмечено, причина появления различных (но эквивалентных) версий регулярных выражений кроется в различных интерпретациях метасимволов, обозначающих символы слова и границу слова, о чем подробнее рассказывается в разделе «Символы слов» в рецепте 2.6.

В последних двух регулярных выражениях используются отдельные категории Юникода, соответствующие буквам, комбинационным знакам (необходимы для сопоставления со словами во многих языках), десятичным цифрам и знакам-соединителям (символ подчеркивания и подобные ему), чтобы обеспечить их эквивалентность предыдущим регулярным выражениям, опиравшимся на использование метасимволов ‹\w› и ‹\W›.

Каждое повторение несохраняющей группы в первых двух регулярных выражениях из трех совпадает с целым словом, за которым следует ноль или более символов, не являющихся символами слова. Метасимвол ‹\W› (или символьный класс ‹[^\p{L}\p{M}\p{Nd}\p{Pc}]›) внутри группы может повторяться нулевое число раз в случае, если строка заканчивается символом слова. Но поскольку в результате этого последовательность символов, не являющихся символами слова, фактически делается необязательной для всего процесса сопоставления, становится

необходимой проверка границы слова ‹\b› между ‹\w› и ‹\W› (или между ‹[\p{L}\p{M}\p{Nd}\p{Pc}]› и ‹[^\p{L}\p{M}\p{Nd}\p{Pc}]›), чтобы гарантировать, что при каждом повторении группа действительно будет совпадать с целым словом. Без проверки границы слова группа при некотором повторении могла бы совпасть с любой частью слова, а при последующих повторениях – с дополнительными частями.

Третья версия регулярного выражения (где была добавлена поддержка диалектов XRegExp, PCRE и Ruby 1.9) действует несколько иначе. Вместо квантификатора звездочки (ноль или более повторений) в ней используется квантификатор плюс (одно или более повторений) и явно допускается совпадение с нулевым числом символов, только когда процесс сопоставления достигнет конца строки. Это позволяет избежать использования метасимвола границы слова, который был необходим, чтобы обеспечить точность подсчета, так как в диалектах XRegExp, PCRE или Ruby метасимвол ‹\b› не поддерживает Юникод. В диалекте Java метасимвол ‹\b› *поддерживает* Юникод, при том что метасимвол ‹\w› в этом же диалекте его не поддерживает (если не установлен флаг UNICODE_CHARACTER_CLASS в Java 7).

К сожалению, ни одна из этих возможностей не позволяет корректно обрабатывать слова, в которых используются буквы и цифры, не входящие в набор ASCII, в диалектах JavaScript и Ruby 1.8. Одно из возможных решений заключается в том, чтобы переориентировать регулярное выражение на подсчет пробельных символов, а не последовательностей символов слов, как показано ниже:

```
^\s*(?:\S+(?:\s+|$)){10,100}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Perl, PCRE, Python, Ruby

Во многих случаях это выражение будет давать те же результаты, что и предыдущие решения, хотя они не являются точными эквивалентами. Например, одно из различий заключается в том, что составные слова, разделенные дефисом (такие как «far-reaching»), в этом случае будут интерпретироваться не как два слова, а как одно.

См. также

Рецепт 4.8, где демонстрируется, как ограничить ввод не длиной, а определенным набором символов (алфавитно-цифровыми, только символами ASCII и т. д.).

Рецепт 4.10, где объясняются тонкости точного ограничения количества строк в тексте.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.7

рассказывается о сопоставлении с символами Юникода. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

4.10. Ограничение числа строк в тексте

Задача

Необходимо убедиться, что испытуемый текст содержит не больше пяти строк независимо от того, сколько символов содержится в этом тексте.

Решение

Символы или последовательности символов, применяемые для разделения строк, могут отличаться в разных операционных системах в зависимости от принятых в них соглашений, используемых приложений, предпочтений пользователей и т. д. Поэтому для выработки идеального решения необходимо ответить на вопрос о том, какие соглашения об обозначении начала новой строки должны поддерживаться. Решения, следующие ниже, поддерживают соглашения о разрывах строк, принятые в MS-DOS/Windows (‹\r\n›), в устаревшей версии Mac OS (‹\r›) и в UNIX/Linux/BSD/OS X (‹\n›).

Регулярное выражение

Следующие три регулярных выражения для разных диалектов имеют два отличия. В первом регулярном выражении вместо несохраняющей группировки, которая оформляется как ‹(?:···)›, используется атомарная группировка, которая оформляется как ‹(?>···)›, потому что атомарная группировка обладает несколько более высокой эффективностью в диалектах, которые ее поддерживают. Python и JavaScript не поддерживают атомарную группировку, поэтому в решениях для этих диалектов она не используется. Другое отличие заключается в метасимволах, применяемых для проверки начала и конца строки (метасимволы ‹\A› или ‹^› проверяют совпадение с началом строки, а метасимволы ‹\z›, ‹\Z› или ‹$› – с концом строки). Подробнее эти различия будут обсуждаться ниже в этом рецепте. Все три регулярных выражения в точности соответствуют одним и тем же строкам:

```
\A(?>[^\r\n]*(?>\r\n?|\n)){0,4}[^\r\n]*\z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
\A(?:[^\r\n]*(?:\r\n?|\n)){0,4}[^\r\n]*\Z
```

Параметры: нет
Диалект: Python

```
^(?:[^\r\n]*(?:\r\n?|\n)){0,4}[^\r\n]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалект: JavaScript

Пример для PHP (PCRE)

```
if (preg_match('/\A(?>[^\r\n]*(?>\r\n?|\n)){0,4}[^\r\n]*\z/',
               $_POST['subject'])) {
    print 'Испытуемый текст содержит не более пяти строк';
} else {
    print 'Испытуемый текст содержит более пяти строк';
}
```

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.6.

Обсуждение

Все регулярные выражения, продемонстрированные к настоящему моменту в этом рецепте, используют группировку для поиска совпадений с последовательностями, обозначающими разрывы строк, оформленными в стиле MS-DOS/Windows, устаревшей версии Mac OS или UNIX/Linux/BSD/OS X, за которыми следует произвольное число любых других символов. Группировка повторяется от нуля до четырех раз, поскольку согласно условиям требуется обеспечить совпадение не более чем с четырьмя разрывами строк в тексте, содержащем не более пяти строк. После группы допускается наличие еще одной, последней последовательности символов, не содержащей символов разрыва строк, формирующей пятую строку.

Ниже приводится первая версия регулярного выражения, разбитая на отдельные части. Отличия в версиях для других диалектов мы объясним ниже:

```
\A          # Проверить совпадение с началом строки.
(?>         # Группировка, но не сохраняющая
            #     и не запоминающая позицию для возврата
  [^\r\n]*  #   Ноль или более любых символов, кроме CR и LF.
  (?>       #   Группировка, но не сохраняющая
            #     и не запоминающая позицию для возврата
    \r\n?   #     Соответствует символу возврата каретки (CR),
            #      за которым может следовать символ перевода строки LF.
   |        #    Или:
    \n      #     Соответствует символу перевода строки LF
  )         #   Конец несохраняющей атомарной группы.
){0,4}      # Конец группы, повторить ее от нуля до четырех раз.
[^\r\n]*    # Ноль или более любых символов, кроме CR и LF.
\z          # Проверить совпадение с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Первый метасимвол ‹\A› совпадает с позицией начала текста, а символ ‹\z› – с концом. Это позволит гарантировать, что весь текст будет содержать не более пяти строк, потому что, если не привязать регулярное выражение к началу текста, оно может совпасть с любыми пятью строками внутри более объемного текста.

Следующая атомарная группа (рецепт 2.14) включает в себя символьный класс, которому соответствует любое количество символов, не являющихся символами конца строки, и подгруппу, совпадающую с последовательностью символов конца строки. Совпадение с символьным классом объявлено необязательным (следующим за ним квантификатором, допускающим нулевое число совпадений), но подгруппа является обязательной, и ей должна соответствовать точно одна последовательность символов конца строки на каждое повторение внешней группы. За внешней группой следует квантификатор, позволяющий находить от нуля до четырех совпадений с ней. Нулевому числу совпадений соответствует пустой текст или единственная строка, не содержащая символы конца строки.

Вслед за внешней группой следует еще один символьный класс, которому соответствует ноль или более любых символов, не являющихся символами конца строки. Это позволяет регулярному выражению найти совпадение с пятой строкой в испытуемом тексте, если она присутствует. Этот класс нельзя опустить и заменить предшествующий ему квантификатор на ‹{0,5}›, потому что в этом случае для совпадения с регулярным выражением испытуемый текст должен был бы заканчиваться символом завершения строки. Кроме того, если последняя строка окажется пустой, такая замена привела бы к возможности совпадения с текстом, содержащим шесть строк, поскольку шесть строк разделяются пятью разрывами строк. Но это противоречит условиям задачи.

Во всех этих регулярных выражениях подгруппа определяет одну из трех последовательностей символов конца строки:

- Символ возврата каретки, за которым следует символ перевода строки (‹\r\n›, последовательность символов конца строки, обычная для MS-DOS/Windows)
- Отдельный символ возврата каретки (‹\r›, символ конца строки в устаревшей версии Mac OS)
- Отдельный символ перевода строки (‹\n›, символ конца строки, обычный для UNIX/Linux/BSD/OS X)

Теперь перейдем к различиям между диалектами.

Первая версия регулярного выражения (совместимая со всеми диалектами, за исключением Python и JavaScript) использует не простую несохраняющую группировку, а атомарную. В некоторых случаях использование атомарной группировки может оказывать более существенное влияние, но в данной ситуации она просто позволяет механизму регулярных выражений избежать ненужных возвратов, необходимость в которых может возникнуть, если попытка совпадения начнет терпеть неудачу.

Еще одно отличие между версиями для разных диалектов – это метасимволы, используемые для проверки совпадения с началом и концом текста. Все диалекты, обсуждаемые в этой книге, поддерживают метасимволы ‹^› и ‹$›. Тогда почему в некоторых решениях вместо них используются ‹\A›, ‹\Z› и ‹\z›? Если говорить коротко, это объясняется тем, что значения этих метасимволов несколько отличаются в разных диалектах. Для подробного объяснения придется погрузиться в историю развития регулярных выражений...

Когда в языке Perl выполнялось чтение строк из файла, получающаяся строка заканчивалась символом конца строки. Вследствие этого в Perl было «расширено» традиционное значение метасимвола ‹$›, которое затем было скопировано большинством других диалектов регулярных выражений. В дополнение к совпадению с абсолютным концом всего текста метасимвол ‹$› в языке Perl совпадает также с позицией непосредственно перед символом конца строки. Кроме того, в язык Perl были введены еще две проверки, совпадающие с концом текста: ‹\Z› и ‹\z›. Якорный метасимвол ‹\Z› в языке Perl имеет почти то же значение, что и метасимвол ‹$›, за исключением того, что на него не оказывает влияние режим, позволяющий метасимволам ‹^› и ‹$› совпадать с символами конца строки. Метасимвол ‹\z› всегда совпадает только с абсолютным концом текста, без всяких исключений. Поскольку в этом рецепте мы имеем дело с символами конца строки, с помощью которых производится подсчет строк в тексте, то, чтобы убедиться в отсутствии шестой строки, здесь используется проверка ‹\z› в диалектах, поддерживающих ее.

Большинство других диалектов регулярных выражений скопировало из языка Perl якорные метасимволы, обозначающие конец строки/текста. Диалекты .NET, Java, PCRE и Ruby поддерживают оба метасимвола, ‹\Z› и ‹\z›, которые в этих диалектах имеют то же значение, что и в языке Perl. В Python имеется только метасимвол ‹\Z› (в верхнем регистре), но он имеет немного сбивающее с толку значение, совпадая только с абсолютным концом текста, как метасимвол ‹\z› (в нижнем регистре) в языке Perl. В диалекте JavaScript отсутствуют якорные метасимволы «z», но, в отличие от всех остальных диалектов, рассматриваемых в этой книге, якорный метасимвол ‹$› в этом диалекте совпадает только с абсолютным концом текста (когда отключен режим, допускающий совпадение метасимволов ‹^› и ‹$› с границами строк).

В случае с метасимволом ‹\A› ситуация немного лучше. Он всегда совпадает только с началом текста и одинаково интерпретируется во всех диалектах, рассматриваемых в этой книге, за исключением JavaScript (который не поддерживает его).

Несмотря на существование досадных разногласий между диалектами, одно из преимуществ регулярных выражений, которые приводятся в этой книге, состоит в том, что в большинстве случаев вам не придется беспокоиться по поводу этих различий. Кровавые по-

дробности, подобные только что описанным, мы включаем только для тех, кто желает получить более глубокие знания.

Варианты

Работа с необычными разделителями строк

Регулярные выражения, продемонстрированные выше, ограничиваются поддержкой последовательностей символов конца строки, обычных для MS-DOS/Windows, устаревшей версии Mac OS и UNIX/Linux/BSD/OS X. Однако существуют некоторые другие редко встречающиеся завершающие пробельные символы, с которыми можно столкнуться. Следующие регулярные выражения учитывают возможность появления этих дополнительных символов, также ограничивая совпадение не более чем пятью строками.

```
\A(?>\V*\R){0,4}\V*\z
```

Параметры: нет
Диалекты: PCRE 7.2 (с параметром PCRE_BSR_UNICODE), Perl 5.10

```
\A(?>[^\n-\r\x85\x{2028}\x{2029}]*(?>\r\n?|↵
[\n-\f\x85\x{2028}\x{2029}])){0,4}[^\n-\r\x85\x{2028}\x{2029}]*\z
```

Параметры: нет
Диалекты: Java 7, PCRE, Perl

```
\A(?>[^\n-\r\u0085\u2028\u2029]*(?>\r\n?|↵
[\n-\f\u0085\u2028\u2029])){0,4}[^\n-\r\u0085\u2028\u2029]*\z
```

Параметры: нет
Диалекты: .NET, Java, Ruby 1.9

```
\A(?>[^\n-\r\x85\u2028\u2029]*(?>\r\n?|↵
[\n-\f\x85\u2028\u2029])){0,4}[^\n-\r\x85\u2028\u2029]*\z
```

Параметры: нет
Диалекты: .NET, Java

```
\A(?:[^\n-\r\x85\u2028\u2029]*(?:\r\n?|↵
[\n-\f\x85\u2028\u2029])){0,4}[^\n-\r\x85\u2028\u2029]*\Z
```

Параметры: нет
Диалект: Python

```
^(?:[^\n-\r\x85\u2028\u2029]*(?:\r\n?|↵
[\n-\f\x85\u2028\u2029])){0,4}[^\n-\r\x85\u2028\u2029]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалект: JavaScript

Ruby 1.8 не поддерживает символы Юникода в регулярных выражениях, и потому не может использовать ни один из использованных здесь приемов. Ruby 1.9 не поддерживает сокращенный синтаксис ‹\x*NN*› для представления не-ASCII символов (с кодами выше 0x7F), и потому в решении для него приходится использовать ‹\u0085› вместо ‹\x85›.

Все эти регулярные выражения обрабатывают разделители, перечисленные в табл. 4.1 вместе с их кодовыми пунктами и именами в Юникоде. В этот список включены символы, которые стандартом Юникода определяются как символы завершения строк.

Таблица 4.1. Разделители строк

Последовательность в Юникоде	Эквивалент в регулярных выражениях	Название	Аббревиатура	Когда используется
U+000D U+000A	‹\r\n›	Возврат каретки и перевод строки	CRLF	Текстовые файлы в Windows и MS-DOS
U+000A	‹\n›	Перевод строки	LF	Текстовые файлы в UNIX, Linux, BSD и OS X
U+000B	‹\v› или ‹\x0B›	Вертикальная табуляция	VT	(Редко)
U+000C	‹\f›	Перевод формата	FF	(Редко)
U+000D	‹\r›	Возврат каретки	CR	Текстовые файлы в старых версиях Mac OS
U+0085	‹\x85› или ‹\x0085›	Следующая строка	NEL	Текстовые файлы в суперкомпьютерах IBM
U+2028	‹\u2028› или ‹\x{2028}›	Разделитель строк	LS	(Редко)
U+2029	‹\u2029› или ‹\x{2029}›	Разделитель абзацев	PS	(Редко)

См. также

Рецепт 4.9, где демонстрируется, как ограничить уже не набор символов, а длину текста.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 демонстрируется, как организовать сопоставление с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.7 рассказывается о сопоставлении с символами Юникода. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.14 описывается атомарная группировка.

4.11. Проверка утвердительных ответов

Задача

Необходимо проверить параметры настройки или ответ, введенный в командной строке, на совпадение с положительным утверждением. Требуется обеспечить достаточную гибкость в принятии ответов, чтобы такие ответы, как `true`, `t`, `yes`, `y`, `okay`, `ok` и `1`, принимались в любых комбинациях символов верхнего и нижнего регистров.

Решение

Использовать регулярное выражение, объединяющее в себе все допустимые формы и позволяющее выполнить проверку с помощью одного простого критерия.

Регулярное выражение

```
^(?:1|t(?:rue)?|y(?:es)?|ok(?:ay)?)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пример для JavaScript

```
var yes = /^(?:1|t(?:rue)?|y(?:es)?|ok(?:ay)?)$/i;

if (yes.test(subject)) {
    alert("Yes");
} else {
    alert("No");
}
```

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.6. В рецепте 3.4 демонстрируется, как устанавливать параметры регулярных выражений, включая модификатор «нечувствительности к регистру символов».

Обсуждение

Ниже приводится регулярное выражение, разложенное на отдельные составляющие. Комбинации элементов, которые проще читать вместе, приводятся в одной строке:

```
^           # Проверка совпадения с началом строки.
(?:         # Группировка, но несохраняющая...
  1         #   Проверка соответствия литералу "1".
 |          #  или...
  t(?:rue)? #   Проверка соответствия с "t",
            #   за которым может следовать "rue".
 |          #  или...
```

```
  y(?:es)?  #   Проверка соответствия с "y",
            #   за которым может следовать "es".
 |          #  или...
  ok(?:ay)? #   Проверка соответствия с "ok",
            #   за которым может следовать "ay".
)           # Конец несохраняющей группы.
$           # Проверка совпадения с концом строки.
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования

Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Регулярное выражение фактически проверяет совпадение с одним из семи литералов без учета регистра символов. Его можно записать разными способами. Например, выражение ‹^(?:[1ty]|true|yes|ok(?:ay)?)$› ничуть не хуже предыдущего. Простая проверка совпадения с одной из семи альтернатив, например ‹^(?:1|t|true|y|yes|ok|okay)$›, также прекрасно справляется с поставленной задачей, хотя с точки зрения производительности обычно лучше минимизировать количество альтернатив, перечисляемых с помощью оператора выбора ‹|› в символьных классах и необязательных элементах (к которым применяется квантификатор ‹?›). В данном случае падение производительности наверняка не будет превышать нескольких микросекунд, но для пользы дела всегда следует помнить о проблемах, связанных с производительностью регулярных выражений. Иногда различия между этими способами реализации могут вызывать неподдельное удивление.

Во всех этих примерах потенциальные совпадения заключены в несохраняющую группу, чтобы ограничить область действия операторов выбора. Если убрать группировку и использовать, например, такое выражение, как ‹^true|yes$›, механизм регулярных выражений будет искать «начало строки, за которым следует фрагмент «true» или «yes», за которым следует конец строки». Выражение ‹^(?:true|yes)$› предписывает механизму регулярных выражений отыскать начало строки, потом отыскать «true» или «yes», а потом – конец строки.

См. также

Рецепты 5.2 и 5.3, где приводятся дополнительные примеры сопоставления с любым из множества слов.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.12. Проверка номеров социального страхования

Задача

Необходимо проверить, является ли некоторый текст допустимым номером социального страхования.

Решение

Если требуется просто убедиться, что текст следует формату записи номеров социального страхования, и отвергнуть очевидно недопустимые значения, легко решить эту задачу можно с помощью приведенного ниже регулярного выражения. Если требуется более строгое решение, которое выполняет сверку с данными Управления социального обеспечения, чтобы определить, принадлежит ли этот номер существующему человеку, обращайтесь к разделу «См. также» этого рецепта.

Регулярное выражение

```
^(?!000|666)[0-8][0-9]{2}-(?!00)[0-9]{2}-(?!0000)[0-9]{4}$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пример для Python

```
if re.match(r"^(?!000|666)[0-8][0-9]{2}-(?!00)[0-9]{2}-↵
(?!0000)[0-9]{4}$", sys.argv[1]):
    print "Корректный номер социального страхования"
else:
    print "Некорректный номер социального страхования"
```

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.6.

Обсуждение

Номера социального страхования в Соединенных Штатах состоят из девяти цифр и имеют формат *AAA-GG-SSSS*:

- Первые три цифры прежде (до 2011 года) определяли географический регион и называются *номером зоны*. Номер зоны не может иметь значения 000, 666 и в диапазоне от 900 до 999.
- Четвертая и пятая цифры называются *номером группы*, который может изменяться в диапазоне от 01 до 99.
- Последние четыре цифры составляют *личный номер*, который может изменяться в диапазоне от 0001 до 9999.

Решение для этого рецепта соблюдает все ограничения, перечисленные выше. Ниже снова приводится регулярное выражение, на этот раз разложенное на отдельные составляющие:

```
^                # Проверка совпадения с началом строки.
(?!000|666)      # Гарантировать отсутствие здесь совпадения с "000" или "666".
[0-8]            # Соответствует цифре в диапазоне от 0 до 8.
[0-9]{2}         # Соответствует цифре точно два раза.
-                # Соответствует литералу "-".
(?!00)           # Гарантировать отсутствие здесь совпадения с "00".
[0-9]{2}         # Соответствует цифре точно два раза.
-                # Соответствует литералу "-".
(?!0000)         # Гарантировать отсутствие здесь совпадения с "0000".
[0-9]{4}         # Соответствует цифре точно четыре раза.
$                # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Помимо метасимволов ‹^› и ‹$›, выполняющих проверку совпадения с началом и концом строки, это регулярное выражение можно разбить на три группы цифр, разделенных дефисом. Первая группа допускает совпадение с любыми числами в диапазоне от 000 до 899, но предшествующая ей негативная опережающая проверка ‹(?!000|666)› препятствует совпадению с последовательностями цифр «000» и «666». Это ограничение можно реализовать и без опережающей проверки, но она существенно упрощает конструкцию регулярного выражения. Если вам потребуется исключить возможность совпадения с последовательностями «000» и «666» без применения опережающей проверки, тогда подвыражение ‹(?!000|666)[0-8][0-9]{2}› необходимо будет заменить подвыражением ‹(?:00[1-9]|0[1-9][0-9]|[1-578][0-9]{2}|6[0-57-9][0-9]|66[0-57-9])›. В этом менее удобочитаемом решении используется несколько сопоставлений с числовыми диапазонами, о которых рассказывается в рецепте 6.7.

Вторая и третья группы просто совпадают с двух- и четырехзначными числами соответственно, но используют негативную опережающую проверку, чтобы обеспечить невозможность совпадения со значениями, состоящими из одних нулей.

Варианты

Поиск номеров социального страхования в документах

Если требуется отыскать номера социального страхования в большом документе или в длинной строке ввода, следует заменить якорные метасимволы ‹^› и ‹$› метасимволами границы слова. Механизмы регулярных выражений интерпретируют все алфавитно-цифровые символы и символ подчеркивания как символы слова.

```
\b(?!000|666)[0-8][0-9]{2}-(?!00)[0-9]{2}-(?!0000)[0-9]{4}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

На сайте Управления социального обеспечения действует служба проверки номеров социального страхования (Social Security Number Verification Service, SSNVS) по адресу *http://www.socialsecurity.gov/employer/ssnv.htm*, которая предоставляет два способа убедиться, что имя и номер социального страхования соответствуют записи в Управлении социального обеспечения.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

4.13. Проверка номеров ISBN

Задача

Необходимо проверить корректность международного стандартного номера книги (International Standard Book Number, ISBN), который может быть записан в старом (ISBN-10) или в текущем (ISBN-13) формате. Требуется обеспечить возможность разделения идентификатора ISBN и других частей номера символом дефиса или пробела. Все следующие примеры являются допустимыми номерами ISBN:

- ISBN 978-0-596-52068-7
- ISBN-13: 978-0-596-52068-7
- 978 0 596 52068 7
- 9780596520687
- ISBN-10 0-596-52068-9
- 0-596-52068-9.

Решение

Невозможно выполнить полную проверку номера ISBN исключительно средствами регулярного выражения, потому что последняя цифра вычисляется с применением алгоритма получения контрольной суммы.

Регулярные выражения

Три решения, которые приводятся в этом разделе, позволяют проверить соответствие одному из двух форматов – ISBN 10 или ISBN 13 – либо соответствие им обоим. Каждое решение сначала приводится в простом виде, а затем в режиме свободного форматирования с коммента-

риями. Диалект JavaScript не поддерживает режим свободного форматирования, но при использовании других языков программирования вы можете выбрать то решение, которое лучше соответствует вашим желаниям.

В решениях, оформленных в режиме свободного форматирования, символы пробелов экранированы обратными слэшами. Диалект Java требует, чтобы пробелы экранировались даже в символьных классах.

ISBN-10:

```
^(?:ISBN(?:-10)?:?␣)?(?=[0-9X]{10}$|(?=(?:[0-9]+[-␣]){3})[-␣0-9X]{13}$)↵
[0-9]{1,5}[-␣]?[0-9]+[-␣]?[0-9]+[-␣]?[0-9X]$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^
(?:ISBN(?:-10)?:?\ )?     # Необязательный идентификатор ISBN/ISBN-10
(?=                       # Предварительная проверка формата
                          #    (опережающая проверка):
  [0-9X]{10}$             #   Требуется 10 цифр/X-ов (без разделителей).
 |                        #  Или:
  (?=(?:[0-9]+[-\ ]){3}) #   Требуется 3 разделителя
  [-\ 0-9X]{13}$          #     на 13 символов.
)                         # Конец предварительной проверки формата.
[0-9]{1,5}[-\ ]?          # Идентификатор группы (от 1 до 5 цифр).
[0-9]+[-\ ]?[0-9]+[-\ ]? # Идентификаторы издателя и книги.
[0-9X]                    # Цифра контрольной суммы.
$
```

Параметры: режим свободного форматирования

Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

ISBN-13:

```
^(?:ISBN(?:-13)?:?␣)?(?=[0-9]{13}$|(?=(?:[0-9]+[-␣]){4})[-␣0-9]{17}$)↵
97[89][-␣]?[0-9]{1,5}[-␣]?[0-9]+[-␣]?[0-9]+[-␣]?[0-9]$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^
(?:ISBN(?:-13)?:?\ )?     # Необязательный идентификатор ISBN/ISBN-13.
(?=                       # Предварительная проверка формата
                          #    (опережающая проверка):
  [0-9]{13}$              #   Требуется 13 цифр (без разделителей).
 |                        #  Или:
  (?=(?:[0-9]+[-\ ]){4}) #   Требуется 4 разделителя
  [-\ 0-9]{17}$           #     на 17 символов.
)                         # Конец предварительной проверки формата.
97[89][-\ ]?              # Префикс ISBN-13.
[0-9]{1,5}[-\ ]?          # Идентификатор группы (от 1 до 5 цифр).
[0-9]+[-\ ]?[0-9]+[-\ ]? # Идентификаторы издателя и книги.
[0-9]                     # Цифра контрольной суммы.
$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

ISBN-10 или ISBN-13:

```
^(?:ISBN(?:-1[03])?:?␣)?(?=[0-9X]{10}$|(?=(?:[0-9]+[-␣]){3})↵
[-␣0-9X]{13}$|97[89][0-9]{10}$|(?=(?:[0-9]+[-␣]){4})[-␣0-9]{17}$)↵
(?:97[89][-␣]?)?[0-9]{1,5}[-␣]?[0-9]+[-␣]?[0-9]+[-␣]?[0-9X]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^
(?:ISBN(?:-1[03])?:?\ )? # Необязательный идентификатор ISBN/ISBN-10/ISBN-13.
(?=                      # Проверка формата (опережающая проверка):
  [0-9X]{10}$            #   Требуется 10 цифр/X-ов (без разделителей).
 |                       #  Или:
  (?=(?:[0-9]+[-\ ]){3}) #   Требуется 3 разделителя
  [-\ 0-9X]{13}$         #     на 13 символов.
 |                       #  Или:
  97[89][0-9]{10}$       #   978/979 плюс 10 цифр (всего 13).
 |                       #  Или:
  (?=(?:[0-9]+[-\ ]){4}) #   Требуется 4 разделителя
  [-\ 0-9]{17}$          #     на 17 символов.
)                        # Конец предварительной проверки формата.
(?:97[89][-\ ]?)?        # Необязательный префикс ISBN-13.
[0-9]{1,5}[-\ ]?         # Идентификатор группы (от 1 до 5 цифр).
[0-9]+[-\ ]?[0-9]+[-\ ]? # Идентификаторы издателя и книги.
[0-9X]                   # Цифра контрольной суммы.
$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Пример для JavaScript с проверкой контрольной суммы

```
var subject = document.getElementById("isbn").value;

// регулярное выражение проверяет соответствие формату ISBN-10 или ISBN-13
var regex = /^(?:ISBN(?:-1[03])?:? )?(?=[0-9X]{10}$|↵
(?=(?:[0-9]+[- ]){3})[- 0-9X]{13}$|97[89][0-9]{10}$|↵
(?=(?:[0-9]+[- ]){4})[- 0-9]{17}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?↵
[0-9]+[- ]?[0-9]+[- ]?[0-9X]$/;

if (regex.test(subject)) {
    // Удалить цифры, не имеющие отношение к ISBN, и разбить номер на части,
    // представив их в виде массива
    var chars = subject.replace(/[- ]|^ISBN(?:-1[03])?:?/g, "").split("");
    // Удалить последнюю цифру ISBN из массива 'chars'
    // и присвоить ее переменной 'last'
    var last = chars.pop();
    var sum = 0;
    var check, i;
```

```
if (chars.length == 9) {
    // Вычислить контрольную цифру ISBN-10
    chars.reverse();
    for (i = 0; i < chars.length; i++) {
        sum += (i + 2) * parseInt(chars[i], 10);
    }
    check = 11 - (sum % 11);
    if (check == 10) {
        check = "X";
    } else if (check == 11) {
        check = "0";
    }
} else {
    // Вычислить контрольную цифру ISBN-13
    for (i = 0; i < chars.length; i++) {
        sum += (i % 2 * 2 + 1) * parseInt(chars[i], 10);
    }
    check = 10 - (sum % 10);
    if (check == 10) {
        check = "0";
    }
}

if (check == last) {
    alert("Корректный номер ISBN");
    } else {
        alert("Некорректная контрольная цифра номера ISBN");
    }
} else {
    alert("Некорректный номер ISBN");
}
```

Пример для Python с проверкой контрольной суммы

```
import re
import sys

subject = sys.argv[1]

# регулярное выражение проверяет соответствие формату ISBN-10 или ISBN-13
regex = re.compile("^(?:ISBN(?:-1[03])?:? )?(?=[0-9X]{10}$|↵
(?=(?:[0-9]+[- ]){3})[- 0-9X]{13}$|97[89][0-9]{10}$|↵
(?=(?:[0-9]+[- ]){4})[- 0-9]{17}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?↵
[0-9]+[- ]?[0-9]+[- ]?[0-9X]$")

if regex.search(subject):
    # Удалить цифры, не имеющие отношение к ISBN, и разбить номер на части,
    # представив их в виде массива
    chars = list(re.sub("[- ]|^ISBN(?:-1[03])?:?", "", subject))
    # Удалить последнюю цифру ISBN из массива 'chars'
    # и присвоить ее переменной 'last'
    last = chars.pop()
```

```
if len(chars) == 9:
    # Вычислить контрольную цифру ISBN-10
    val = sum((x + 2) * int(y) for x,y in enumerate(reversed(chars)))
    check = 11 - (val % 11)
    if check == 10:
        check = "X"
    elif check == 11:
        check = "0"
else:
    # Вычислить контрольную цифру ISBN-13
    val = sum((x % 2 * 2 + 1) * int(y) for x,y in enumerate(chars))
    check = 10 - (val % 10)
    if check == 10:
        check = "0"

if (str(check) == last):
    print "Корректный номер ISBN"
else:
    print "Некорректная контрольная цифра номера ISBN"
else:
    print "Некорректный номер ISBN"
```

Обсуждение

ISBN – это уникальный идентификатор продаваемых книг и подобных им продуктов. 10-значный формат ISBN был опубликован как международный стандарт ISO 2108 в 1970 году. Все номера ISBN, которые присваиваются начиная с 1 января 2007 года, состоят из 13 цифр.

Номера в форматах ISBN-10 и ISBN-13 делятся на четыре и на пять элементов соответственно. Три элемента имеют переменную длину, а остальные один или два элемента – фиксированную. Обычно все части отделяются друг от друга дефисами или пробелами. Ниже приводится краткое описание каждого элемента:

- 13-значные номера ISBN начинаются с префикса 978 или 979.
- *Идентификатор группы* определяет группу стран, говорящих на одном языке. Этот идентификатор может иметь длину от одной до пяти цифр.
- *Идентификатор издательства* имеет переменную длину и присваивается национальным агентством ISBN.
- *Идентификатор книги* также имеет переменную длину и выбирается издателем.
- Заключительный символ называется *контрольной цифрой* и вычисляется с применением алгоритма получения контрольной суммы. В формате ISBN-10 контрольная цифра может быть числом от 0 до 9 или символом X (число 10 в римской системе записи чисел), тогда как в формате ISBN-13 контрольная цифра может быть только числом в диапазоне от 0 до 9. Так как для разных форматов ISBN ис-

пользуются разные алгоритмы вычисления контрольной цифры, допустимые символы в них отличаются.

Все три решения, представленные выше, содержат схожие части, поэтому мы сосредоточимся на регулярном выражении из решения «ISBN-10 или ISBN-13», разложенное на отдельные составляющие. Первое подвыражение `‹^(?:ISBN(?:-1[03])?:?␣)?›` содержит три необязательных элемента, что позволяет ему совпадать с любым из семи фрагментов, следующих ниже (в конце всех фрагментов, за исключением варианта с пустой строкой, имеется пробел):

- `ISBN␣`
- `ISBN-10␣`
- `ISBN-13␣`
- `ISBN:␣`
- `ISBN-10:␣`
- `ISBN-13:␣`
- *Пустая строка (префикс ISBN отсутствует)*

Вслед за первым подвыражением `‹^(?:ISBN(?:-1[03])?:?␣)?›` следует позитивная опережающая проверка, гарантирующая соблюдение одного из четырех ограничений (разделенных оператором выбора `‹|›`), определяющих длину и допустимый набор символов для остальной части совпадения, а также количество разделителей (ноль или три для ISBN-10, ноль или четыре для ISBN-13). Из-за наличия четырех вариантов выбора опережающая проверка получилась достаточно длинной. Вот как она выглядит: `‹(?=[0-9X]{10}$|(?=(?:[0-9]+[-␣]){3})[-␣0-9X]{13}$|97[89][0-9]{10}$|(?=(?:[0-9]+[-␣]){4})[-␣0-9]{17}$)›`. Так как в ней целиком разобраться трудно, ниже приводится подробное описание каждого из вариантов по отдельности. Все четыре ограничения оканчиваются якорным метасимволом `‹$›`, гарантирующим отсутствие последующего текста, не укладывающегося ни в один из шаблонов:

`‹[0-9X]{10}$›`

формат ISBN-10 без разделителей (всего 10 символов)

`‹(?=(?:[0-9]+[-␣]){3})[-␣0-9X]{13}$›`

формат ISBN-10 с тремя разделителями (всего 13 символов)

`‹97[89][0-9]{10}$›`

формат ISBN-13 без разделителей (всего 13 символов)

`‹(?=(?:[0-9]+[-␣]){4})[-␣0-9]{17}$›`

формат ISBN-13 с четырьмя разделителями (всего 17 символов)

Два из этих вариантов (которые допускают наличие разделителей) включают собственные опережающие проверки, позволяющие гарантировать корректное число разделителей, прежде чем перейти к проверке общей длины строки.

После того как позитивная опережающая проверка проверит длину и набор символов, можно произвести сопоставление отдельных элементов номера ISBN, не беспокоясь об их суммарной длине. Подвыражению `‹(?:97[89][-]?)?›` соответствует префикс «978» или «979», обязательный в формате ISBN-13. Совпадение с несохраняющей группой сделано необязательным, потому что она не обнаружит соответствия в испытуемой строке, имеющей формат ISBN-10. Подвыражению `‹[0-9]{1,5}[-]?›` соответствуют от одной до пяти цифр идентификатора группы и необязательный разделитель, возможно следующий за ними. Подвыражению `‹[0-9]+[-]?[0-9]+[-]?›` соответствуют идентификаторы издательства и книги, имеющие переменную длину, и их необязательные разделители. Наконец, подвыражению `‹[0-9X]$›` соответствует контрольная цифра в конце строки.

Несмотря на то что регулярное выражение в состоянии убедиться, что в качестве последней цифры используется корректный символ (цифра или символ X), оно не сможет определить, представляет ли эта цифра корректную контрольную сумму ISBN. Для обеспечения гарантии, что цифры номера ISBN были введены без ошибок и не являются случайным набором цифр, используется один из двух алгоритмов вычисления контрольной суммы (в зависимости от того, в каком формате записан номер: ISBN-10 или ISBN-13). Реализации обоих алгоритмов показаны в примерах программного кода на языках JavaScript и Python. В следующих разделах описываются правила вычисления контрольных сумм, чтобы вы могли реализовать их в других языках программирования.

Контрольная сумма ISBN-10

Контрольная сумма для номеров в формате ISBN-10 может принимать значения в диапазоне от 0 до 10 (значение 10 записывается римской цифрой X). Она вычисляется следующим образом:

1. Каждая из первых 9 цифр умножается на число, убывающее в порядке от 10 до 2, а результаты умножения суммируются.
2. Сумма делится на 11.
3. Остаток деления (не частное) вычитается из числа 11.
4. Если результат равен 11, в качестве контрольной суммы используется число 0; для представления результата, равного 10, используется символ X.

Ниже приводится пример вычисления контрольной цифры для номера `0-596-52068-?` в формате ISBN-10:

```
Шаг 1:
sum = 10×0 + 9×5 + 8×9 + 7×6 + 6×5 + 5×2 + 4×0 + 3×6 + 2×8
    =    0 +  45  + 72 +  42 +  30 +  10 +   0 +  18 +  16
    = 233
Шаг 2:
    233 ÷ 11 = 21, остаток 2
Шаг 3:
```

```
    11 - 2 = 9
Шаг 4:
    9 [подстановка иного значения не требуется]
```

Была получена контрольная цифра 9, поэтому полный номер имеет вид: ISBN 0-596-52068-9.

Контрольная сумма ISBN-13

Контрольная сумма для номеров в формате ISBN-13 может принимать значения в диапазоне от 0 до 9 и вычисляется похожим способом:

1. По мере перемещения по номеру вправо каждая из первых 12 цифр умножается на 1 или на 3, а результаты умножения суммируются.
2. Сумма делится на 10.
3. Остаток деления (не частное) вычитается из числа 10.
4. Если результат равен 10, в качестве контрольной суммы используется число 0.

Ниже приводится пример вычисления контрольной цифры для номера 978-0-596-52068-? в формате ISBN-13:

```
Шаг 1:
sum = 1×9 + 3×7 + 1×8 + 3×0 + 1×5 + 3×9 + 1×6 + 3×5 + 1×2 + 3×0 + 1×6 + 3×8
    =   9 +  21 +   8 +   0 +   5 +  27 +   6 +  15 +   2 +   0 +   6 +  24
    = 123
Шаг 2:
    123 ÷ 10 = 12, остаток 3
Шаг 3:
    10 - 3 = 7
Шаг 4:
    7 [подстановка иного значения не требуется]
```

Была получена контрольная цифра 7, поэтому полный номер имеет вид: ISBN 978-0-596-52068-7.

Варианты

Поиск номеров ISBN в документах

В этой версии регулярного выражения «ISBN-10 или ISBN-13» якорные метасимволы заменены метасимволом границы слова, чтобы его можно было использовать для поиска отдельных номеров ISBN в объемном тексте. Кроме того, в этой версии идентификатор «ISBN» сделан обязательным по двум причинам. Во-первых, обязательное наличие идентификатора позволит избежать ложных совпадений (без этого требования регулярное выражение может совпадать с 10- и 13-значными числами), а во-вторых, идентификатор ISBN официально является обязательным при выводе на печать:

```
\bISBN(?:-1[03])?:?*(?=[0-9X]{10}$|(?=(?:[0-9]+[-*]){3})↵
[-*0-9X]{13}$|97[89][0-9]{10}$|(?=(?:[0-9]+[-*]){4})[-*0-9]{17}$)↵
```

```
(?:97[89][-*]?)?[0-9]{1,5}[-*]?[0-9]+[-*]?[0-9]+[-*]?[0-9X]\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пропуск некорректных идентификаторов ISBN

Предыдущие регулярные выражения допускают совпадение с номером ISBN-10, которому предшествует идентификатор «ISBN-13», и наоборот. В следующем регулярном выражении используется условная конструкция (рецепт 2.17), чтобы убедиться, что за идентификаторами «ISBN-10» и «ISBN-13» следует номер ISBN соответствующего типа. Оно допускает совпадение с любым из форматов ISBN-10 и ISBN-13, если тип не указан явно. В большинстве случаев это регулярное выражение будет чрезмерно избыточным, потому что те же результаты можно получить, последовательно используя более простые регулярные выражения, отдельные для ISBN-10 и ISBN-13, приведенные выше. Оно представлено здесь исключительно для демонстрации возможностей регулярных выражений:

```
^
(?:ISBN(-1(?:(0)|3))?:?\ )?
(?(1)
  (?(2)
    # ISBN-10
    (?=[0-9X]{10}$|(?=(?:[0-9]+[- ]){3})[- 0-9X]{13}$)
    [0-9]{1,5}[- ]?[0-9]+[- ]?[0-9]+[- ]?[0-9X]
   |
    # ISBN-13
    (?=[0-9]{13}$|(?=(?:[0-9]+[- ]){4})[- 0-9]{17}$)
    97[89][- ]?[0-9]{1,5}[- ]?[0-9]+[- ]?[0-9]+[- ]?[0-9]
  )
 |
  # Идентификатор не указан явно; допускается ISBN-10 или ISBN-13
  (?=[0-9X]{10}$|(?=(?:[0-9]+[- ]){3})[- 0-9X]{13}$|97[89][0-9]{10}$|
    (?=(?:[0-9]+[- ]){4})[- 0-9]{17}$)
  (?:97[89][- ]?)?[0-9]{1,5}[- ]?[0-9]+[- ]?[0-9]+[- ]?[0-9X]
)
$
```

Параметры: режим свободного форматирования
Диалекты: .NET, PCRE, Perl, Python

См. также

Самую последнюю версию руководства пользователя «ISBN User's Manual» вместе с инструментами проверки отдельных номеров ISBN и преобразования между форматами ISBN-10 и ISBN-13 можно найти на сайте Международного агентства ISBN *http://www.isbn-international.org*.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецеп-

те 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках. В рецепте 2.17 демонстрируется применение условных конструкций.

4.14. Проверка почтовых индексов

Задача

Необходимо выполнить проверку ZIP-кода (почтовый индекс в США), который может записываться в двух форматах: как пятизначное число или как девятизначное число (называется ZIP + 4). Регулярное выражение должно совпадать со строками 12345 и 12345-6789 и не совпадать со строками 1234, 123456, 123456789 или 1234-56789.

Решение

Регулярное выражение

```
^[0-9]{5}(?:-[0-9]{4})?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пример для VB.NET

```
If Regex.IsMatch(subjectString, "^[0-9]{5}(?:-[0-9]{4})?$") Then
    Console.WriteLine("Допустимый ZIP-код")
Else
    Console.WriteLine("Недопустимый ZIP-код")
End If
```

Помощь в реализации этого регулярного выражения на других языках программирования можно получить в рецепте 3.6.

Обсуждение

Ниже приводится регулярное выражение проверки ZIP-кода, разложенное на отдельные составляющие:

```
^          # Проверка совпадения с началом строки.
[0-9]{5}   # Соответствует цифре точно пять раз.
(?:        # Группировка, но несохраняющая:
  -        #   Соответствует литералу "-".
  [0-9]{4} # Соответствует цифре точно четыре раза.
)          # Конец несохраняющей группы.
  ?        # Повторить предшествующую группу ноль или один раз.
$          # Проверка совпадения с концом строки.
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Это регулярное выражение весьма очевидное, поэтому нам нечего добавить к его описанию. Чтобы использовать его для поиска ZIP-кодов внутри более длинной строки, следует заменить якорные метасимволы ‹^› и ‹$› метасимволом границы слова, что в результате даст выражение ‹\b[0-9]{5}(?:-[0-9]{4})?\b›.

Существует один допустимый код ZIP+4, который не совпадает с данным регулярным выражением: 10022-SHOE. Это единственный ZIP-код, включающий буквы. В 2007 он был специально присвоен обувному магазину Saks на 50-й авеню в Нью-Йорке. Больше никаких подобных ZIP-кодов почтовая служба США не создавала, по крайней мере, до сих пор. Почтовые отправления с индексом 10022 все еще будут попадать в этот обувной магазин (и этот индекс будет проходить все проверки), поэтому вряд ли стоит изменять регулярное выражение из-за этого единственного исключения.

См. также

Для тех, кому приходится посылать почтовые отправления за пределы США, в рецепте 4.15 приводится решение для работы с почтовыми индексами в Канаде, а в рецепте 4.16 – с почтовыми индексами в Великобритании.

В рецепте 4.17 демонстрируется поиск номеров почтовых ящиков в адресах на случай, если вам потребуется обрабатывать эти номера отдельно от обычных адресов с названиями улиц.

Найти город по ZIP-коду, определить ZIP-коды для города и штата или адреса можно на сайте *https://www.usps.com/zip4/*. Но в действительности ZIP-коды определяют маршруты доставки, а не конкретные географические точки, поэтому существует множество необычных ситуаций, когда ZIP-коды пересекают границы штатов или соответствуют военным кораблям, отдельным зданиям крупных компаний или почтовым ящикам.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.15. Проверка почтовых индексов, используемых в Канаде

Задача

Необходимо проверить, является ли строка почтовым индексом, используемым в Канаде.

Решение

```
^(?!.*[DFIOQU])[A-VXY][0-9][A-Z]*?[0-9][A-Z][0-9]$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Негативная опережающая проверка, находящаяся в начале этого регулярного выражения, предотвращает появление в испытуемой строке символов D, F, I, O, Q или U. Символьный класс `[A-VXY]`, следующий далее, исключает возможность появления символа W или Z в начале индекса. Кроме этих двух исключений, в канадских почтовых индексах просто используется чередующаяся последовательность из шести алфавитных и цифровых символов с пробелом в середине. Например, регулярное выражение будет совпадать с последовательностью K1A 0B1, которая является почтовым индексом центрального почтового отделения города Оттава, Канада.

См. также

В рецепте 4.14 описывается решение задачи проверки почтовых индексов США, а в рецепте 4.16 – почтовых индексов Великобритании.

В рецепте 4.17 демонстрируется поиск номеров почтовых ящиков в адресах на случай, если вам потребуется обрабатывать эти номера отдельно от обычных адресов с названиями улиц.

Почтовая служба Канады предоставляет веб-страницу для определения почтового индекса по адресу *http://www.canadapost.ca/cpotools/apps/fpc/personal/findByCity*.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

4.16. Проверка почтовых индексов, используемых в Великобритании

Задача

Необходимо написать регулярное выражение, которое будет совпадать с почтовым индексом, используемым в Великобритании.

Решение

```
^[A-Z]{1,2}[0-9R][0-9A-Z]?*[0-9][ABD-HJLNP-UW-Z]{2}$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Почтовые индексы в Великобритании могут содержать от пяти до семи алфавитно-цифровых символов, разделенных пробелом. Правила, определяющие, какие символы и в каких позициях могут появляться, отличаются большой сложностью и имеют множество исключений. Данное регулярное выражение реализует основные из этих правил.

Для тех, кому требуется регулярное выражение, реализующее все правила и учитывающее все исключения ценой потери удобочитаемости, оно приводится ниже:

```
^(?:(?:[A-PR-UWYZ][0-9]{1,2}|[A-PR-UWYZ][A-HK-Y][0-9]{1,2}↵
|[A-PR-UWYZ][0-9][A-HJKSTUW]|[A-PR-UWYZ][A-HK-Y][0-9]↵
[ABEHMNPRV-Y])*[0-9][ABD-HJLNP-UW-Z]{2}|GIR 0AA)$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Британский стандарт BS7666 по адресу *http://interim.cabinetoffice.gov.uk/govtalk/schemasstandards/e-gif/datastandards/address/postcode.aspx*, где описываются правила составления почтовых индексов.

На веб-сайте Королевской почты *http://www.royalmail.com/postcode-finder* предоставляется возможность определения почтового индекса по адресу.

В рецептах 4.14 и 4.15 описывается решение задач проверки почтовых индексов США и Канады.

В рецепте 4.17 демонстрируется поиск номеров почтовых ящиков в адресах.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.17. Поиск адресов, содержащих номер почтового ящика

Задача

Необходимо отыскать адреса, содержащие номер почтового ящика, и предупредить пользователя о том, что информация о доставке должна содержать адрес улицы.

Решение

Регулярное выражение

```
^(?:Post(?:al)?\s(?:Office\s)?|P[.\s]?O\.?\s)?Box\b
```

Параметры: нечувствительность к регистру, символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пример для C#

```
Regex regexObj = new Regex(
    @"^(?:Post(?:al)? (?:Office )?|P[. ]?O\.? )?Box\b",
    RegexOptions.IgnoreCase | RegexOptions.Multiline
);
if (regexObj.IsMatch(subjectString) {
    Console.WriteLine("Строка не выглядит как адрес улицы");
} else {
    Console.WriteLine("Можно отправлять");
}
```

Помощь в применении этого регулярного выражения на других языках программирования можно получить в рецепте 3.5. В рецепте 3.4 демонстрируется, как устанавливать параметры регулярных выражений, упомянутые здесь.

Обсуждение

Следующее объяснение записано в режиме свободного форматирования, поэтому каждый значимый символ пробела в регулярном выражении экранирован символом обратного слэша:

```
^                   # Проверка совпадения с началом строки.
(?:                 # Группировка, но несохраняющая...
  Post(?:al)?\      #   Соответствует тексту "Post " или "Postal ".
  (?:Office\ )?     #   Необязательное соответствие тексту "Office ".
 |                  #  Или:
  P[.\ ]?           #   Соответствует "P" и, возможно, запятой или пробелу.
  O\.?\             # Соответствует "O" и, возможно, точке и пробелу.
```

```
)?              # Сделать совпадение с группой необязательным.
Box             # Соответствует тексту "Box".
\b              # Проверить совпадение с границей слова.
```

Параметры: нечувствительность к регистру, символам ^ и $ соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Регулярному выражению соответствуют все примеры строк, следующие ниже, если они находятся в начале строки:

- Post Office Box
- Postal Box
- post box
- P.O. box
- P O Box
- Po. box
- PO Box
- Box

Несмотря на все предпринятые меры предосторожности, в реальной жизни можно столкнуться с ситуацией, когда почтовое отправление доставляется не по тому адресу или когда при указанном верном адресе отправление возвращается обратно, потому что многие люди достаточно вольно трактуют адреса.

Чтобы уменьшить этот риск, лучше заранее предупредить, что начинать адрес с указания почтового ящика недопустимо. Если будет получено совпадение с этим регулярным выражением, можно предупредить пользователя, что был введен номер почтового ящика, но не было указано, как до него добраться.

См. также

В рецептах 4.14, 4.15 и 4.16 описывается решение задач проверки почтовых индексов США, Канады и Великобритании соответственно.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.18. Преобразование из формата «имя фамилия» в формат «фамилия, имя»

Задача

Требуется преобразовать имена людей из формата «имя фамилия» в формат «фамилия, имя» для составления списка, отсортированного по фамилиям в алфавитном порядке. При этом необходимо принять во внимание другие элементы, составляющие полное имя, например преобразовать полные имена из формата «имя отчество приставка фамилия дополнение» в формат «фамилия, имя отчество приставка дополнение».

Решение

К сожалению, с помощью регулярных выражений невозможно произвести надежный парсинг имен. Регулярные выражения задают жесткие условия совпадения, тогда как имена людей могут быть настолько необычными, что даже человек может понимать их неправильно. Для определения структуры имен или способов сортировки в алфавитном порядке часто требуется принимать во внимание национальные традиции и даже личные предпочтения. Однако если есть возможность сделать определенные предположения относительно обрабатываемых данных и свести количество ошибок до приемлемого уровня, регулярные выражения помогут найти быстрое решение.

При составлении следующего регулярного выражения мы не пытались реализовать обработку крайних случаев и преднамеренно сохранили его простым.

Регулярное выражение

```
^(.+?)•([^\s,]+)(,?•(?:[JS]r\.?|III?|IV))?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
$2,•$1$3
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\2,•\1\3
```

Диалекты замещающего текста: Python, Ruby

Пример для JavaScript

```
function formatName (name) {
    return name.replace(/^(.+?) ([^\s,]+)(,? (?:[JS]r\.?|III?|IV))?$/i,
                        "$2, $1$3");
}
```

Помощь в применении этого регулярного выражения для поиска и замены в программах на других языках программирования можно получить в рецепте 3.15. В рецепте 3.4 демонстрируется, как устанавливать параметры регулярных выражений, включая модификатор «нечувствительности к регистру символов».

Обсуждение

Для начала рассмотрим это регулярное выражение по частям. Комментарии, поясняющие, какие части имени соответствуют тем или иным сегментам регулярного выражения, будут даны ниже. Так как здесь регулярное выражение записано в режиме свободного форматирования, литералы пробелов экранированы символами обратного слэша:

```
^             # Проверка совпадения с началом строки.
(             # Сохранение совпадения для обратной ссылки 1:
  .+?         #   Соответствует одному или более символам,
              #   минимальное число раз.
)             # Конец сохраняющей группы.
\             # Соответствует литералу пробела.
(             # Сохранение совпадения для обратной ссылки 2:
  [^\s,]+     #   Соответствует одному или более непробельным символам
              #   или запятым.
)             # Конец сохраняющей группы.
(             #   Сохранение совпадения для обратной ссылки 3:
  ,?\         #   Соответствует ", " или " ".
  (?:         #   Группировка, но несохраняющая:
    [JS]r\.? #     Соответствует "Jr", "Jr.", "Sr" или "Sr.".
   |          #    Или:
    III?      #     Соответствует "II" или "III".
   |          #    Или:
    IV        # Соответствует "IV".
  )           # Конец несохраняющей группы.
)?            # Сделать совпадение с группой необязательным.
$             # Проверка совпадения с концом строки.
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Это регулярное выражение реализовано с учетом следующих предположений об испытуемых данных:

- Испытуемая строка содержит, по меньшей мере, имя и фамилию (остальные части полного имени считаются необязательными).
- Имя записано перед фамилией (у некоторых народов принят обратный порядок следования).
- Дополнение к имени, если оно присутствует, имеет одно из следующих значений: «Jr», «Jr.», «Sr», «Sr.», «II», «III» или «IV» с необязательной предшествующей запятой.

В этом регулярном выражении не решены некоторые проблемы, которые стоит рассмотреть:

- Регулярное выражение не в состоянии определять составные фамилии, в которых не используется символ дефиса. Например, полное имя Sacha Baron Cohen будет преобразовано в Cohen, Sacha Baron, тогда как правильное представление Baron Cohen, Sacha.
- Оно не сохраняет приставки перед фамилией, хотя иногда этого требуют национальные традиции или личные предпочтения (например, правильное представление имени «Charles de Gaulle» (Шарль де Голль) имеет вид «de Gaulle, Charles» согласно 16-му изданию «Chicago Manual of Style», который не совпадает с представлением в биографическом словаре Мерриам-Вебстер (Merriam-Webster's Biographical Dictionary)).
- Поскольку якорные метасимволы ‹^› и ‹$› привязывают совпадение к началу и концу строки, никаких замещений не может быть выполнено, если весь испытуемый текст не соответствует шаблону. Следовательно, если совпадение не будет обнаружено (например, когда испытуемый текст содержит одно только имя), имя останется без изменений.

Что касается работы регулярного выражения, оно делит полное имя на части, используя три сохраняющие группы. Затем с помощью обратных ссылок в замещающем тексте эти части собираются в требуемом порядке. В первой сохраняющей группе используется максимально гибкий шаблон ‹.+?› для извлечения имени вместе с любым количеством вторых имен и приставок к фамилии, таких как немецкое «von» или французское «de». Эти части полного имени рассматриваются как единое целое, поскольку они последовательно перечисляются в строке результата. Кроме того, объединение первого и второго имен помогает избежать ошибок, потому что регулярное выражение не в состоянии отличать составные имена, такие как «Mary Lou» или «Norma Jeane», от имени, за которым следует второе имя. Даже люди не всегда в состоянии визуально отличать их.

Подвыражению ‹[^\s,]+› во второй сохраняющей группе соответствует фамилия. Гибкость этого символьного класса, подобно точке в первой сохраняющей группе, позволяет ему совпадать с любыми акцентированными и любыми другими символами, не входящими в набор Latin.

Третьей сохраняющей группе соответствует необязательное дополнение, такое как «Jr.» или «III», из предопределенного списка допустимых значений. Дополнение обрабатывается отдельно от фамилии, потому что оно должно находиться в конце преобразованной строки с полным именем.

Вернемся на мгновение к первой сохраняющей группе. Почему за точкой в первой сохраняющей группе следует минимальный квантификатор ‹+?›, тогда как за символьным классом во второй сохраняющей группе следует максимальный квантификатор ‹+›? Если бы первая группа

(которая обрабатывает имя переменной длины и потому должна продвинуться по строке с полным именем как можно дальше) использовала максимальный квантификатор, то третья сохраняющая группа (которой соответствует дополнение) вообще не получила бы возможность участвовать в сопоставлении. Точка из первой группы могла бы совпасть со всеми символами до самого конца строки, а поскольку третья группа является необязательной, механизм регулярных выражений выполнял бы возврат только для того, чтобы обеспечить совпадение со второй группой, после чего объявил бы об успехе. Вторая группа может использовать максимальный квантификатор, потому что она содержит более ограниченный символьный класс, позволяющий группе совпадать только с одним элементом имени.

В табл. 4.2 приводятся некоторые примеры форматирования имен с помощью данного регулярного выражения и строки замещающего текста.

Таблица 4.2. Форматированные имена

Оригинал	Результат
Robert Downey, Jr.	Downey, Robert, Jr.
John F. Kennedy	Kennedy, John F.
Scarlett O'Hara	O'Hara, Scarlett
Pepé Le Pew	Pew, Pepé Le
J.R.R. Tolkien	Tolkien, J.R.R.
Catherine Zeta-Jones	Zeta-Jones, Catherine

Варианты

Список приставок к фамилии в начале полного имени

Добавив небольшой фрагмент, как показано в следующем регулярном выражении, можно обеспечить вывод перед фамилией приставок к фамилии из предопределенного списка. Конкретно это регулярное выражение распознает следующие значения: «de», «du», «la», «le», «St», «St.», «Ste», «Ste.», «van» и «von». Допускаются любые комбинации из этих значений (например, «de la»):

```
^(.+?)•((?:(?:d[eu]|l[ae]|Ste?\.?|v[ao]n)•)*[^\s,]+)↵
(,?•(?:[JS]r\.?|III?|IV))?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
$2,•$1$3
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\2,•\1\3
```

Диалекты замещающего текста: Python, Ruby

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.13 объясняется, как воздействуют на механизм возвратов максимальные и минимальные квантификаторы. В рецепте 2.21 демонстрируется, как вставлять текст, совпавший с сохраняющей группой, в текст замены.

4.19. Проверка сложности пароля

Задача

Требуется проверить пароль, выбранный пользователем вашего сайта, на соответствие минимальным требованиям к сложности, принятым в вашей организации.

Решение

Следующие регулярные выражения проверяют множество отдельных условий, которые могут компоноваться по мере необходимости для проверки соответствия принятым требованиям. В конце этого раздела приводится несколько примеров программного кода на языке JavaScript, демонстрирующих приемы встраивания описываемых здесь регулярных выражений в процедуру проверки сложности пароля.

Длина от 8 до 32 символов

```
^.{8,32}$
```

Параметры: точке соответствуют границы строк (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

В стандартном диалекте JavaScript отсутствует режим «символам ^ и $ соответствуют границы строк». Используйте в JavaScript ‹[\s\S]› вместо точки, чтобы гарантировать корректную работу регулярного выражения даже с необычными паролями, включающими символы разрыва строк:

```
^[\s\S]{8,32}$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Только видимые символы ASCII и пробелы

Если пароль соответствует следующему регулярному выражению, можно быть уверенными, что он включает только символы A-Z, a-z, 0-9, пробелы и знаки препинания из таблицы символов ASCII. Это выражение не допускает присутствия любых управляющих символов, символов разрыва строк или символов, не входящих в набор ASCII:

```
^[\x20-\x7E]+$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если дополнительно потребуется исключить пробелы, используйте выражение ‹^[\x21-\x7E]+$›.

Один или более алфавитных символов в верхнем регистре

Только символы ASCII в верхнем регистре:

```
[A-Z]
```

Параметры: нет (режим «нечувствительность к регистру символов» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Любые алфавитные символы Юникода в верхнем регистре:

```
\p{Lu}
```

Параметры: нет (режим «нечувствительность к регистру символов» должен быть выключен)
Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Если потребуется проверить наличие хотя бы одного алфавитного символа (необязательно в верхнем регистре), включите режим нечувствительности к регистру или используйте ‹[A-Za-z]›. Для проверки символов Юникода можно использовать категорию Юникода ‹\p{L}›, которая соответствует любой букве из любого алфавита.

Один или более алфавитных символов в нижнем регистре

Только символы ASCII в нижнем регистре:

```
[a-z]
```

Параметры: нет (режим «нечувствительность к регистру символов» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Любые алфавитные символы Юникода в нижнем регистре:

```
\p{Ll}
```

Параметры: нет (режим «нечувствительность к регистру символов» должен быть выключен)

Диалекты: .NET, Java, PCRE, Perl, Ruby 1.9

Одна или более цифр

```
[0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Один или более специальных символов

Только пробелы и знаки препинания ASCII:

```
[ *!"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Любые символы, кроме букв и цифр ASCII:

```
[^A-Za-z0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Запрет появления трех и более идентичных символов, следующих подряд

Следующее регулярное выражение предназначено для исключения паролей вида 111111. Оно должно применяться иначе, чем другие регулярные выражения в этом рецепте. При обнаружении совпадения пароль должен быть признан не соответствующим условиям. Иными словами, регулярное выражение совпадает только со строками, содержащими три и более одинаковых символа, следующих подряд.

```
(.)\1\1
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
([\s\S])\1\1
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Пример простого решения на JavaScript

Следующий программный код объединяет пять требований к паролям:

- Длина от 8 до 32 символов
- Один или более алфавитных символов в верхнем регистре
- Один или более алфавитных символов в нижнем регистре
- Одна или более цифр
- Один или более специальных символов (знаков препинания ASCII или пробелов).

```
function validate(password) {
    var minMaxLength = /^[\s\S]{8,32}$/,
        upper = /[A-Z]/,
        lower = /[a-z]/,
        number = /[0-9]/,
        special = /[ !"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~]/;

    if (minMaxLength.test(password) &&
        upper.test(password) &&
        lower.test(password) &&
        number.test(password) &&
        special.test(password)
    ) {
        return true;
    }
    return false;
}
```

Функция `validate`**, представленная выше, возвращает** `true`**, если переданная ей строка соответствует предъявляемым требованиям. Иначе она возвращает** `false`**.**

Пример решения на JavaScript с подсчетом числа удовлетворяемых требований

Следующий пример требует обязательного соблюдения требования «длина от 8 до 32 символов» и соблюдения не менее трех требований из четырех перечисленных:

- **Один или более алфавитных символов в верхнем регистре**
- **Один или более алфавитных символов в нижнем регистре**
- **Одна или более цифр**
- **Один или более специальных символов (отличных от букв и цифр ASCII).**

```
function validate(password) {
    var minMaxLength = /^[\s\S]{8,32}$/,
        upper = /[A-Z]/,
        lower = /[a-z]/,
        number = /[0-9]/,
        special = /[^A-Za-z0-9]/,
        count = 0;

    if (minMaxLength.test(password)) {
        // Достаточно соблюдения трех правил из четырех
        if (upper.test(password)) count++;
        if (lower.test(password)) count++;
        if (number.test(password)) count++;
        if (special.test(password)) count++;
    }
    return count >= 3;
}
```

Как и прежде, эта версия функции `validate` возвращает `true`, если переданная ей строка соответствует предъявляемым требованиям. Иначе она возвращает `false`.

Пример решения на JavaScript с определением оценки надежности пароля

Этот заключительный пример является самым сложным из представленных. Он присваивает положительное или отрицательное число очков разным условиям и использует регулярные выражения, описанные выше, для вычисления общей оценки пароля. Функция `rankPassword` возвращает число в диапазоне от 0 до 4, соответствующее оценкам пароля: «Too Short» (Слишком короткий), «Weak» (Ненадежный), «Medium» (Средний), «Strong» (Надежный) и «Very Strong» (Очень надежный):

```
var rank = {
    TOO_SHORT: 0,
    WEAK: 1,
    MEDIUM: 2,
    STRONG: 3,
    VERY_STRONG: 4
};

function rankPassword(password) {
    var upper = /[A-Z]/,
    lower = /[a-z]/,
    number = /[0-9]/,
    special = /[^A-Za-z0-9]/,
    minLength = 8,
    score = 0;

if (password.length < minLength) {
    return rank.TOO_SHORT; // Завершить оценку раньше
}

// Увеличить значение score при соблюдении любого из следующих условий
if (upper.test(password)) score++;
if (lower.test(password)) score++;
if (number.test(password)) score++;
if (special.test(password)) score++;

// Уменьшить score, если соблюдено менее трех условий
if (score < 3) score--;

if (password.length > minLength) {
    // Увеличить score на 1 за каждые 2 символа сверх минимальной длины
    score += Math.floor((password.length - minLength) / 2);
}

// Вернуть оценку, опираясь на вычисленное значение score
if (score < 3) return rank.WEAK; // score имеет значение 2 или ниже
```

```
    if (score < 4) return rank.MEDIUM; // score имеет значение 3
    if (score < 6) return rank.STRONG; // score имеет значение 4 или 5
    return rank.VERY_STRONG; // score имеет значение 6 или выше
}

// Проверка...
var result = rankPassword("password1"),
    labels = ["Too Short", "Weak", "Medium", "Strong", "Very Strong"];

alert(labels[result]); // -> Weak (Ненадежный)
```

Благодаря особенностям алгоритма оценки он одинаково хорошо может служить двум целям. Во-первых, его можно использовать, чтобы сообщать пользователю о качестве пароля прямо в процессе ввода. Во-вторых, с его помощью можно отвергать пароли с оценкой ниже требуемого уровня. Например, для отклонения паролей с оценками ниже «Strong» (Надежный) и «Very Strong» (Очень надежный) можно использовать условие `if(result <= rank.MEDIUM)`.

Обсуждение

Пользователи часто склонны выбирать простые или легко запоминающиеся пароли. Но легко запоминаемый пароль не всегда может достаточно надежно защитить доступ к учетной записи и к коммерческой информации компании. Поэтому обычно бывает необходимо защитить пользователей от самих себя и вынудить их придумывать пароли, отвечающие некоторым минимальным требованиям. Однако в разных компаниях и системах могут предъявляться разные требования, и именно поэтому в данный рецепт включен набор регулярных выражений, которые могут служить исходными компонентами для реализации различных комбинаций требований.

Ограничение каждого регулярного выражения проверкой соответствия единственному требованию упрощает их реализацию. Как результат, все регулярные выражения, представленные выше, чрезвычайно просты. Ниже приводятся дополнительные замечания к каждому из них:

Длина от 8 до 32 символов

Чтобы установить иные ограничения минимальной и максимальной длины, следует изменить нижнюю и верхнюю границы в квантификаторе `‹{8,32}›`. Если максимальную длину не требуется ограничивать, используйте квантификатор `‹{8,}›` или удалите якорный метасимвол `‹$›` и измените квантификатор на `‹{8}›`.

Все языки программирования, рассматриваемые в этой книге, поддерживают простые и эффективные способы определения длины строки. Однако применение регулярного выражения дает возможность одновременно проверять минимальную и максимальную длину и сводит его внедрение в реализацию требований к сложности пароля к простому выбору из списка регулярных выражений.

Только видимые символы ASCII и пробелы

Как отмечалось выше, данное регулярное выражение допускает присутствие в испытуемой строке только символов A–Z, a–z, 0–9, пробела и знаков препинания из таблицы символов ASCII. Если говорить более конкретно, к допустимым знакам препинания относятся: !, ", #, $, %, &, ', (,), *, +, -, ., /, :, ;, <, =, >, ?, @, [, \,], ^, _, `, {, |, }, ~ и запятая. Иными словами, все знаки препинания, которые можно ввести с помощью клавиатуры со стандартной раскладкой U.S.

Ограничение паролей указанным набором символов может помочь избежать проблем с кодировками символов, но имейте в виду, что это также ограничивает потенциальную сложность паролей.

Символы в верхнем регистре

Для проверки наличия в пароле двух или более алфавитных символов в верхнем регистре можно использовать регулярное выражение ‹[A-Z].*[A-Z]›. Трех или более — ‹[A-Z].*[A-Z].*[A-Z]› или ‹(?:[A-Z].*){3}›. Если допускается присутствие алфавитных символов Юникода в верхнем регистре, достаточно заменить ‹[A-Z]› в предыдущих примерах на ‹\p{Lu}›. В JavaScript необходимо заменить точки на ‹[\s\S]›.

Символы в нижнем регистре

Как и в регулярном выражении, проверяющем присутствие символов в верхнем регистре, проверить наличие хотя бы двух алфавитных символов в нижнем регистре можно с помощью выражения ‹[a-z].*[a-z]›. Трех или более — ‹[a-z].*[a-z].*[a-z]› или ‹(?:[a-z].*){3}›. Если допускается присутствие алфавитных символов Юникода в нижнем регистре, достаточно заменить каждое вхождение ‹[a-z]› на ‹\p{Ll}›. В JavaScript необходимо заменить точки на ‹[\s\S]›.

Цифры

Проверить наличие в пароле двух или более цифр можно с помощью выражения ‹[0-9].*[0-9]›, а трех или более — ‹[0-9].*[0-9].*[0-9]› или ‹(?:[0-9].*){3}›. В JavaScript необходимо заменить точки на ‹[\s\S]›.

Мы не включили в регулярные выражения сопоставление с десятичными цифрами Юникода (‹\p{Nd}›), потому что в качестве цифр редко используются символы, отличные от символов 0–9 (хотя читатели, говорящие на арабском или хинди, могут с нами не согласиться!).

Специальные символы

Тот же подход, что применялся для проверки присутствия букв и цифр, можно использовать и для проверки присутствия специальных символов. Например, с помощью регулярного выражения ‹[^A-Za-z0-9].*[^A-Za-z0-9]› можно потребовать присутствия в пароле не менее двух специальных символов.

Имейте в виду, что действие символьного класса ‹[^A-Za-z0-9]› отличается от действия сокращения ‹\W› (инвертированной версии сокращения ‹\w›, совпадающего с символом слова). Сокращение ‹\W› совпа-

дает с более узким диапазоном символов, чем ‹[^A-Za-z0-9]›, дополнительно исключая символ подчеркивания, чего в данном случае желательно избежать. В некоторых диалектах регулярных выражений сокращение ‹\W› также исключает любые буквы и десятичные цифры Юникода.

Запрет появления трех и более идентичных символов, следующих подряд

Это регулярное выражение обеспечивает совпадение с повторяющимися символами с помощью обратных ссылок. О том, как действуют обратные ссылки, рассказывается в рецепте 2.10. Если потребуется исключить *любое* число повторяющихся символов, используйте выражение ‹(.)\1›. Чтобы разрешить появление до трех повторяющихся символов, но не более, используйте выражение ‹(.)\1\1\1› или ‹(.)\1{3}›.

Помните, что данное регулярное выражение проверяет условие, обратное предъявляемому требованию, – оно *не должно* совпадать с испытуемым текстом. Совпадение свидетельствует, что в тексте присутствуют повторяющиеся символы, следующие подряд.

Примеры решений на JavaScript

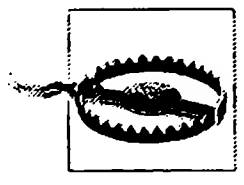
В рецепте приводится три примера на JavaScript, в каждом из которых представленные регулярные выражения используются немного по-разному.

Первый пример требует соответствия всем условиям. Во втором примере успешной считается проверка, если пароль соответствует трем из четырех условий. Третий пример, озаглавленный «Пример решения на JavaScript с определением оценки надежности пароля», является, пожалуй, самым интересным. В нем представлена функция `rankPassword`, суть которой точно соответствует ее названию – она оценивает надежность пароля. С ее помощью можно сообщать пользователям оценки введенных ими паролей и тем самым поощрять выбор более надежных паролей.

Алгоритм, реализованный в функции `rankPassword`, увеличивает и уменьшает внутреннюю оценку пароля исходя из соответствия множеству условий. Если длина пароля меньше указанного минимума (восемь символов), функция немедленно возвращает управление с числовым эквивалентом оценки «Too Short» (Слишком короткий). Несоответствие пароля как минимум трем требованиям снижает оценку на один пункт, но это снижение можно компенсировать увеличением длины пароля, так как каждые два дополнительных символа сверх минимума (восемь символов) увеличивают оценку на один пункт.

Разумеется, исходный код можно изменять и улучшать, приводя его в соответствие с вашими требованиями. Однако и в текущем виде он достаточно хорош независимо от особенностей области его применения. Для оценки мы проверили с его помощью несколько сотен наиболее типичных (и потому ненадежных) паролей, выбираемых пользователями.

Все они получили оценку «Too Short» (Слишком короткий) или «Weak» (Ненадежный) в полном соответствии с нашими ожиданиями.



Использование JavaScript для проверки паролей в веб-браузере может служить дополнительным удобством для пользователей, но не забудьте также реализовать подобную процедуру на сервере. Если этого не сделать, пользователи смогут обойти эту проверку, отключив поддержку JavaScript в браузере или воспользовавшись собственными сценариями.

Варианты

Проверка нескольких требований в одном регулярном выражении

До настоящего момента общая проверка пароля разбивалась на проверки отдельных требований, которые можно выполнить с помощью простых регулярных выражений. Обычно такое решение является лучшим: регулярные выражения остаются более читаемыми и намного проще определить причину несоответствия пароля и вывести сообщение с ее описанием. Более того, такой подход может даже помочь вычислить оценку сложности пароля, как было показано выше. Однако встречаются ситуации, когда все эти преимущества неважны или когда регулярное выражение – это единственное, что доступно. В любом случае, иметь единственное регулярное выражение, проверяющее соответствие пароля сразу нескольким требованиям, – вполне типичное желание, поэтому далее мы посмотрим, как этого добиться. Мы будем проверять соответствие следующим требованиям:

- Длина от 8 до 32 символов
- Один или более алфавитных символов в верхнем регистре
- Один или более алфавитных символов в нижнем регистре
- Одна или более цифр.

Все эти требования проверяются следующим регулярным выражением:

```
^(?=.{8,32}$)(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9]).*
```

Параметры: точке соответствуют границы строк (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

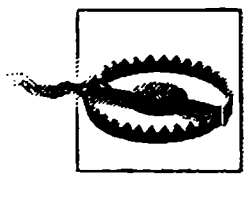
Это регулярное выражение можно использовать со стандартным диалектом JavaScript (не поддерживающим режим «точке соответствуют границы строк»), если заменить каждую из пяти точек символьным классом ‹[\s\S]›. В противном случае выражение будет отвергать некоторые вполне допустимые пароли, содержащие символы разрыва строк. Но в любом случае регулярное выражение не будет совпадать с недопустимыми паролями.

Обратите внимание, что в этом регулярном выражении все правила помещены в отдельные опережающие проверки, находящиеся в начале выражения. Так как опережающие проверки не поглощают символы при обнаружении совпадений (рецепт 2.16), каждая из них выполняет проверку, начиная с начала испытуемой строки. Если опережающая проверка находит совпадение, осуществляется переход к следующей опережающей проверке, которая начинает поиск совпадения с той же позиции в испытуемой строке. Если попытка сопоставления с какой-нибудь из опережающих проверок потерпит неудачу, сопоставление со всем регулярным выражением будет признано неудачным.

Первая опережающая проверка, `(?=.{8,32}$)`, убеждается, что пароль имеет длину от 8 до 32 символов. Обратите внимание на якорный метасимвол `$`, следующий за квантификатором `{8,32}`, – если убрать его, проверка будет соответствовать и паролям, имеющим длину более 32 символов. Следующие опережающие проверки ищут по очереди: алфавитный символ в верхнем регистре, алфавитный символ в нижнем регистре и цифру. Так как каждая опережающая проверка начинает поиск с начала строки, во всех них используется подвыражение `.*` перед соответствующими символьными классами. Это обеспечивает возможность появления любых символов перед искомым совпадением.

Следуя описанному приему, в единственное регулярное выражение можно добавить любое количество опережающих проверок, если, конечно, обязательным условием является соблюдение всех требований.

Конструкция `.*` в самом конце данного регулярного выражения в действительности не требуется. Однако без нее регулярное выражение будет возвращать совпадение нулевой длины в случае успеха. Завершающая комбинация `.*` позволяет регулярному выражению включить сам пароль в совпадение.



С тем же успехом можно было бы записать это регулярное выражение как `^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9]).{8,32}$`, где проверка длины перемещена в конец, после опережающих проверок. К сожалению, это регулярное выражение будет работать неправильно из-за ошибки в Internet Explorer 5.5–8. Эта ошибка была исправлена в новом механизме регулярных выражений, включенном в состав IE9.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 демонстрируется, как организовать сопоставление с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.7 рассказывается о сопоставлении с символами Юникода. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.12 объясняется, как организовать

сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

4.20. Проверка номеров кредитных карт

Задача

Представьте, что вы получили задание реализовать форму заказа для компании, принимающей оплату кредитными картами. Так как за каждую попытку операции с кредитной картой, даже неудачную, взимается плата, вы решили задействовать регулярное выражение, чтобы заранее отсеивать недопустимые номера кредитных карт.

Кроме того, это создаст дополнительные удобства для клиентов. Регулярное выражение в состоянии обнаруживать очевидные опечатки сразу, как только клиент завершит заполнение веб-формы. Обращение же к службе обработки кредитных карт легко может занять от 10 до 30 секунд.

Решение

Чтобы упростить реализацию, это решение было разбито на две части. В первой части удаляются пробелы и дефисы, а во второй выполняется собственно проверка.

Удаление пробелов и дефисов

Необходимо извлечь номер кредитной карты, введенный клиентом, и сохранить его в переменной. Перед выполнением проверки следует отыскать и удалить пробелы и дефисы. Для этого все совпадения со следующим регулярным выражением следует заменить пустым замещающим текстом:

```
[ -]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В рецепте 3.14 показано, как выполнить эту предварительную замену.

Проверка номера

После удаления пробелов и дефисов можно воспользоваться регулярным выражением, следующим ниже, чтобы проверить введенный номер кредитной карты на соответствие формату любой из шести основных компаний, осуществляющих обслуживание кредитных карт. В нем используются именованные сохранения, позволяющие определить название кредитной компании, услугами которой пользуется клиент:

```
^(?:
(?<visa>4[0-9]{12}(?:[0-9]{3})?) |
```

```
(?<mastercard>5[1-5][0-9]{14}) |
(?<discover>6(?:011|5[0-9]{2})[0-9]{12}) |
(?<amex>3[47][0-9]{13}) |
(?<diners>3(?:0[0-5]|[68][0-9])[0-9]{11}) |
(?<jcb>(?:2131|1800|35[0-9]{3})[0-9]{11})
)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?:
(?P<visa>4[0-9]{12}(?:[0-9]{3})?) |
(?P<mastercard>5[1-5][0-9]{14}) |
(?P<discover>6(?:011|5[0-9]{2})[0-9]{12}) |
(?P<amex>3[47][0-9]{13}) |
(?P<diners>3(?:0[0-5]|[68][0-9])[0-9]{11}) |
(?P<jcb>(?:2131|1800|35[0-9]{3})[0-9]{11})
)$
```

Параметры: режим свободного форматирования
Диалекты: PCRE, Python

Диалекты Java (в версиях с 4 по 6), Perl 5.8 и ниже и Ruby 1.8 не поддерживают именованное сохранение. Поэтому в этих диалектах необходимо использовать нумерованное сохранение. Группа 1 сохраняет номер карты Visa, группа 2 – MasterCard и так далее, вплоть до группы 6 – JCB:

```
^(?:
(4[0-9]{12}(?:[0-9]{3})?) |          # Visa
(5[1-5][0-9]{14}) |                  # MasterCard
(6(?:011|5[0-9]{2})[0-9]{12}) |      # Discover
(3[47][0-9]{13}) |                   # AMEX
(3(?:0[0-5]|[68][0-9])[0-9]{11}) |   # Diners Club
((?:2131|1800|35[0-9]{3})[0-9]{11}) # JCB
)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Стандартный диалект JavaScript не поддерживает режим свободного форматирования. Удалив пробелы и комментарии, мы получим:

```
^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|↵
(6(?:011|5[0-9]{2})[0-9]{12})|(3[47][0-9]{13})|↵
(3(?:0[0-5]|[68][0-9])[0-9]{11})|((?:2131|1800|35[0-9]{3})[0-9]{11}))$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если нет необходимости определять тип кредитной карты, можно убрать шесть сохраняющих групп, окружающих подвыражения для каждого типа карт, так как они служат только этой цели.

Рецепт 3.6 поможет добавить это регулярное выражение в форму заказа, чтобы реализовать проверку номера карты. Если для обработки раз-

ных типов карт используются разные службы или если необходимо сохранить некоторую информацию для статистики, можно воспользоваться рецептом 3.9, чтобы узнать, какой именованной или нумерованной сохраняющей группе принадлежит совпадение. Это поможет узнать название кредитной компании, услугами которой пользуется клиент.

Пример веб-страницы со сценарием JavaScript

```
<html>
<head>
<title>Credit Card Test</title>
</head>

<body>
<h1>Проверка кредитной карты</h1>

<form>
<p>Пожалуйста, введите номер вашей кредитной карты:</p>

<p><input type="text" size="20" name="cardnumber"
  onkeyup="validatecardnumber(this.value)"></p>

<p id="notice">(номер карты не был введен)</p>
</form>

<script>
function validatecardnumber(cardnumber) {
    // Удалить пробелы и дефисы
    cardnumber = cardnumber.replace(/[ -]/g, '');
    // Проверить корректность номера карты
    // Регулярное выражение сохранит номер в одной из сохраняющих групп
    var match = /^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|↵
(6(?:011|5[0-9]{2})[0-9]{12})|(3[47][0-9]{13})|(3(?:0[0-5]|[68][0-9])↵
[0-9]{11})|((?:2131|1800|35[0-9]{3})[0-9]{11}))$/.exec(cardnumber);
    if (match) {
        // Список типов карт в том же порядке, что и в регулярном выражении
        var types = ['Visa', 'MasterCard', 'Discover', 'American Express',
                     'Diners Club', 'JCB'];
        // Отыскать совпавшую сохраняющую группу
        // Пропустить нулевой элемент в массиве совпадений (общее совпадение)
        for (var i = 1; i < match.length; i++) {
            if (match[i]) {
                // Вывести тип карты для данной группы
                document.getElementById('notice').innerHTML = types[i - 1];
                break;
            }
        }
    } else {
        document.getElementById('notice').innerHTML =
            '(ошибка в номере карты)';
    }
```

```
}
</script>
</body>
</html>
```

Обсуждение

Удаление пробелов и дефисов

Цифры рельефного номера на самой кредитной карте обычно делятся на группы, по четыре цифры в каждой. Это упрощает чтение номера. Вполне естественно, что многие пытаются ввести номер карты в таком же виде, включая пробелы.

Написать регулярное выражение, которое будет проверять номер кредитной карты, допуская возможность появления пробелов, дефисов и прочих разделителей, намного сложнее, чем регулярное выражение, которое допускает только появление цифр. Поэтому, чтобы не утруждать клиента, заставляя его повторно вводить номер без пробелов или дефисов, можно просто выполнить операцию поиска с заменой и удалить их, прежде чем приступать к проверке номера и выполнять его передачу службе обработки кредитных карт.

Регулярному выражению ‹[•-]› соответствует символ пробела или дефиса. Замена всех совпадений с этим регулярным выражением пустым текстом обеспечит удаление всех пробелов и дефисов.

Номера кредитных карт могут состоять только из цифр. Поэтому вместо символьного класса ‹[•-]›, позволяющего удалить только пробелы и дефисы, можно воспользоваться метасимволом ‹\D›, что позволит удалить все символы, не являющиеся цифрами.

Проверка номера

Каждая компания, обслуживающая кредитные карты, использует свой формат номера. Мы использовали это обстоятельство, чтобы не заставлять клиента вводить название компании вместе с номером – название компании определяется по номеру. Ниже приводятся описания форматов для каждой компании:

Visa

13 или 16 цифр, номер начинается с 4.

MasterCard

16 цифр, номер начинается с чисел от 51 до 55.

Discover

16 цифр, номер начинается с 6011 или с 65.

American Express

15 цифр, номер начинается с 34 или 37.

Diners Club

14 цифр, номер начинается с чисел от 300 до 305, 36 или 38.

JCB

15 цифр, номер начинается с чисел 2131 или 1800; или 16 цифр, номер начинается с 35.

Если необходимо обеспечить прием кредитных карт только определенных типов, нежелательные типы можно удалить из регулярного выражения. При удалении JCB не забудьте также удалить из регулярного выражения последний оператор ‹|›. Регулярное выражение, оканчивающееся последовательностью ‹||› или ‹|)›, будет интерпретировать пустые строки как допустимые номера кредитных карт.

Например, чтобы обеспечить прием кредитных карт только компаний Visa, MasterCard и AMEX, можно воспользоваться следующим регулярным выражением:

```
^(?:
4[0-9]{12}(?:[0-9]{3})? | # Visa
5[1-5][0-9]{14} |         # MasterCard
3[47][0-9]{13}            # AMEX
)$
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Альтернативное решение:

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|3[47][0-9]{13})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если необходимо отыскать номера кредитных карт в тексте документа, якорные метасимволы следует заменить метасимволом ‹\b› границы слова.

Внедрение решения в веб-страницу

Пример в разделе «Пример веб-страницы со сценарием JavaScript» выше демонстрирует, как можно было бы добавить эти два регулярных выражения в форму заказа. Для поля ввода номера кредитной карты определен обработчик события `onkeyup`, который вызывает функцию `validatecardnumber()`. Эта функция извлекает номер кредитной карты из поля ввода, удаляет из него пробелы и дефисы и выполняет проверку с помощью регулярного выражения с нумерованными сохраняющими группами. Результат проверки отображается путем замещения содержимого последнего абзаца на странице.

Если регулярное выражение не обнаружит совпадения, метод *regexp.*`exec()` вернет значение `null` и на странице будет выведен текст (`ошибка в номере карты`). Если совпадение будет найдено, метод *regexp*.`exec()` вернет массив строк. Нулевой элемент этого массива хранит совпадение со

всем регулярным выражением. Элементы с 1 по 6 хранят фрагменты, совпавшие с соответствующими им сохраняющими группами.

В нашем регулярном выражении присутствует шесть сохраняющих групп, разделенных операторами выбора. Это означает, что только одна группа будет участвовать в совпадении и сохранит номер кредитной карты. Остальные элементы массива будут содержать пустой текст (значение `undefined` или пустую строку в зависимости от используемого браузера). Функция проверяет шесть групп, одну за другой. Когда обнаруживается непустая группа, номер карты опознается и на странице выводится ее тип.

Дополнительная проверка с применением алгоритма Луна

Имеется возможность произвести дополнительную проверку номера кредитной карты, прежде чем приступать к обработке заказа. Последняя цифра в номере кредитной карты является контрольной суммой, вычисляемой в соответствии с *алгоритмом Луна*. Так как этот алгоритм требует выполнения арифметических операций, его невозможно реализовать средствами регулярных выражений.

Проверку с применением алгоритма Луна можно добавить в пример веб-страницы, который приводился в этом рецепте, вставив вызов функции `luhn(cardnumber);` **перед инструкцией** `else` **в функции** `validatecardnumber()`. **В этом случае проверка с применением алгоритма Луна будет выполняться, только когда регулярное выражение будет обнаруживать совпадение и после определения типа кредитной карты. При этом для алгоритма Луна определять тип кредитной карты не требуется. Для всех кредитных карт используется один и тот же метод вычисления контрольной суммы.**

На языке JavaScript алгоритм Луна можно реализовать, как показано ниже:

```
function luhn(cardnumber) {
    // Создать массив с цифрами, составляющими номер кредитной карты
    var digits = cardnumber.split('');
    for (var i = 0; i < digits.length; i++) {
        digits[i] = parseInt(digits[i], 10);
    }
    // Выполнить расчет по алгоритму Луна для массива
    var sum = 0;
    var alt = false;
    for (var i = digits.length - 1; i >= 0; i--) {
        if (alt) {
            digits[i] *= 2;
            if (digits[i] > 9) {
                digits[i] -= 9;
            }
        }
```

```
        sum += digits[i];
        alt = !alt;
    }
    // Сообщить о корректности номера карты
    if (sum % 10 == 0) {
        document.getElementById("notice").innerHTML +=
            '; проверка Луна пройдена';
    } else {
        document.getElementById("notice").innerHTML +=
            '; проверка Луна не пройдена';
    }
}
```

Эта функция принимает строку с номером кредитной карты в виде аргумента. Номер кредитной карты может содержать только цифры. В нашем примере функция `validatecardnumber()` уже удалила пробелы и дефисы и определила, что номер карты содержит допустимое число цифр.

Сначала функция разбивает строку на массив символов, а затем выполняет обход массива, конвертируя символы в числа. Если этого не сделать, в результате в переменной `sum` окажется строка, собранная из цифр, а не сумма чисел.

Фактический алгоритм работает с массивом и вычисляет контрольную сумму. Если остаток от деления контрольной суммы на число 10 равен нулю, номер карты считается верным, в противном случае номер содержит ошибку.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

4.21. Европейские регистрационные номера плательщиков НДС

Задача

Вам дано задание реализовать веб-форму заказа для компании, работающей в Евросоюзе.

Европейские налоговые законы предусматривают следующее: когда компания, имеющая регистрационный номер плательщика НДС (ваш клиент) и размещающаяся в одной из стран Евросоюза, приобретает какой-либо товар у поставщика (ваша компания), размещающегося в лю-

бой другой стране Евросоюза, поставщик не должен закладывать НДС (налог на добавленную стоимость) в стоимость товара. Если покупатель не имеет регистрационного номера плательщика НДС, поставщик должен включить НДС в стоимость и перевести сумму НДС местной налоговой службе. Поставщик обязан использовать регистрационный номер плательщика НДС покупателя как доказательство для налоговой службы, что он не должен перечислять НДС. Это означает, что для поставщика очень важно проверить регистрационный номер плательщика НДС покупателя, прежде чем выполнить сделку, не облагаемую налогом.

Самой часто встречающейся причиной некорректности регистрационных номеров являются обычные опечатки, допущенные клиентом. Чтобы сделать процесс оформления заказа более дружественным и быстрым, необходимо использовать регулярное выражение и проверить регистрационный номер, пока клиент заполняет форму заказа. Реализовать такую проверку можно на стороне клиента, в JavaScript, или на стороне сервера с помощью сценария, принимающего форму заказа. Если номер не соответствует регулярному выражению, клиент сможет тут же исправить опечатку.

Решение

Чтобы упростить реализацию, это решение было разбито на две части. В первой части удаляются пробелы и дефисы, а во второй выполняется собственно проверка.

Удаление пробелов и знаков пунктуации

Необходимо извлечь регистрационный номер плательщика НДС, введенный клиентом, и сохранить его в переменной. Прежде чем выполнять проверку корректности номера, следует заменить пустыми строками все совпадения со следующим регулярным выражением:

```
[-.•]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В рецепте 3.14 показано, как выполнить эту предварительную замену. Мы предполагаем, что клиент не может вводить какие-либо знаки пунктуации, за исключением дефисов, точек и пробелов. Любые другие недопустимые символы будут обнаруживаться при последующей проверке.

Проверка номера

После удаления пробелов и знаков пунктуации можно выполнить проверку регистрационного номера для любой из 27 стран, входящих в Евросоюз, с помощью следующего регулярного выражения:

```
^(
(AT)?U[0-9]{8} |                                # Австрия
```

```
(BE)?0[0-9]{9} |                                  # Бельгия
(BG)?[0-9]{9,10} |                                # Болгария
(CY)?[0-9]{8}L |                                  # Кипр
(CZ)?[0-9]{8,10} |                                # Чешская республика
(DE)?[0-9]{9} |                                   # Германия
(DK)?[0-9]{8} |                                   # Дания
(EE)?[0-9]{9} |                                   # Эстония
(EL|GR)?[0-9]{9} |                                # Греция
(ES)?[0-9A-Z][0-9]{7}[0-9A-Z] |                   # Испания
(FI)?[0-9]{8} |                                   # Финляндия
(FR)?[0-9A-Z]{2}[0-9]{9} |                        # Франция
(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3}) |     # Великобритания
(HU)?[0-9]{8} |                                   # Венгрия
(IE)?[0-9]S[0-9]{5}L |                            # Ирландия
(IT)?[0-9]{11} |                                  # Италия
(LT)?([0-9]{9}|[0-9]{12}) |                       # Литва
(LU)?[0-9]{8} |                                   # Люксембург
(LV)?[0-9]{11} |                                  # Латвия
(MT)?[0-9]{8} |                                   # Мальта
(NL)?[0-9]{9}B[0-9]{2} |                          # Нидерланды
(PL)?[0-9]{10} |                                  # Польша
(PT)?[0-9]{9} |                                   # Португалия
(RO)?[0-9]{2,10} |                                # Румыния
(SE)?[0-9]{12} |                                  # Швеция
(SI)?[0-9]{8} |                                   # Словения
(SK)?[0-9]{10}                                    # Словакия
)$
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

В регулярном выражении, приведенном выше, используется режим свободного форматирования, чтобы упростить его изменение в будущем. Время от времени к Евросоюзу присоединяются новые страны, и страны-участницы меняют свои правила, касающиеся регистрационных номеров плательщиков НДС. К сожалению, в JavaScript не поддерживается режим свободного форматирования. Для этого языка программирования все регулярное выражение следует поместить в одну строку:

```
^((AT)?U[0-9]{8}|(BE)?0[0-9]{9}|(BG)?[0-9]{9,10}|(CY)?[0-9]{8}L|↵
(CZ)?[0-9]{8,10}|(DE)?[0-9]{9}|(DK)?[0-9]{8}|(EE)?[0-9]{9}|↵
(EL|GR)?[0-9]{9}|(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]|(FI)?[0-9]{8}|↵
(FR)?[0-9A-Z]{2}[0-9]{9}|(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})|↵
(HU)?[0-9]{8}|(IE)?[0-9]S[0-9]{5}L|(IT)?[0-9]{11}|↵
(LT)?([0-9]{9}|[0-9]{12})|(LU)?[0-9]{8}|(LV)?[0-9]{11}|(MT)?[0-9]{8}|↵
(NL)?[0-9]{9}B[0-9]{2}|(PL)?[0-9]{10}|(PT)?[0-9]{9}|(RO)?[0-9]{2,10}|↵
(SE)?[0-9]{12}|(SI)?[0-9]{8}|(SK)?[0-9]{10})$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Рецепт 3.6 поможет добавить это регулярное выражение в форму заказа.

Обсуждение

Удаление пробелов и знаков пунктуации

Чтобы упростить чтение регистрационных номеров плательщиков НДС, люди часто при вводе добавляют дополнительные знаки пунктуации, чтобы разбить их на группы цифр. Например, клиент из Германии может ввести свой номер `DE123456789` как `DE 123.456.789`.

Практически невозможно написать единственное регулярное выражение, которому соответствуют любые регистрационные номера для любой из 27 стран, введенные в любом формате. Поскольку знаки пунктуации служат исключительно для повышения удобочитаемости, намного проще будет сначала удалить их и только потом выполнить проверку получившегося регистрационного номера.

Регулярному выражению `‹[-. ]›` соответствует дефис, точка или пробел. Операция замены всех совпадений с этим регулярным выражением пустой строкой удалит знаки пунктуации, часто используемые при вводе регистрационных номеров плательщиков НДС.

Регистрационные номера состоят только из букв и цифр. Поэтому вместо выражения `‹[-. ]›` для удаления любых недопустимых символов можно использовать выражение `‹[^A-Z0-9]›`.

Проверка номера

Оба регулярных выражения, проверяющие регистрационный номер, являются идентичными. Единственное отличие состоит в том, что в первом используется режим свободного форматирования, что делает его более удобочитаемым, и в нем присутствуют комментарии, указывающие на страны. Диалект JavaScript (если не используется библиотека XRegExp) не поддерживает режим свободного форматирования, тогда как другие диалекты предоставляют такую возможность.

Чтобы обеспечить совпадение с регистрационным номером для любой из 27 стран Евросоюза, в регулярном выражении используется оператор выбора. Форматы номеров имеют вид, как показано в табл. 4.3.

Строго говоря, двухбуквенные обозначения стран являются частью регистрационного номера. Однако люди часто опускают их, так как название страны уже указывается в счете. Регулярное выражение будет принимать регистрационные номера и без кода страны. Если необходимо, чтобы код страны вводился в обязательном порядке, следует убрать из регулярного выражения все знаки вопроса. В этом случае при выводе сообщения об ошибке необходимо известить клиента, что код страны требуется указывать в обязательном порядке.

Если предполагается принимать заказы только из определенных стран, можно убрать из регулярного выражения те страны, которые не разре-

шено выбрать в форме заказа. Удаляя альтернативы из регулярного выражения, не забывайте удалять и операторы выбора ‹|›, отделяющие удаляемую альтернативу от предыдущей или последующей альтернативы. Если этого не сделать, в регулярном выражении окажутся два оператора выбора, следующие друг за другом ‹||›. Последовательность ‹||› добавит в регулярное выражение альтернативу, которой соответствует пустая строка, сделав возможной ситуацию, когда форма заказа будет воспринимать пустой регистрационный номер, как корректный.

Таблица 4.3. Форматы номеров плательщиков НДС стран Евросоюза

Страна	Формат номера плательщика НДС
Австрия	U*99999999*
Бельгия	*0999999999*
Болгария	*999999999* или *9999999999*
Кипр	*99999999*L
Чешская республика	*99999999*, *999999999* или *9999999999*
Германия	*999999999*
Дания	*99999999*
Эстония	*999999999*
Греция	*999999999*
Испания	X*9999999*X
Финляндия	*99999999*
Франция	XX*999999999*
Великобритания	*999999999*, *999999999999* или XX*999*
Венгрия	*99999999*
Ирландия	*9*S*99999*L
Италия	*9999999999*
Литва	*999999999* или *9999999999*
Люксембург	*99999999*
Латвия	*9999999999*
Мальта	*99999999*
Нидерланды	*999999999*B*99*
Польша	*999999999*
Португалия	*999999999*
Румыния	*99*, *999*, *9999*, *99999*, *999999*, *9999999*, *99999999*, *999999999* или *9999999999*
Швеция	*9999999999*
Словения	*99999999*
Словакия	*999999999*

Все 27 альтернатив сгруппированы в одну группу. Эта группа помещена между символом крышки и знаком доллара, которые привязывают совпадение с регулярным выражением к началу и к концу испытуемой строки. Вследствие этого вся введенная строка будет проверяться как регистрационный номер.

Если возникнет необходимость отыскивать регистрационные номера в тексте документа, якорные метасимволы следует заменить метасимволом ‹\b› границы слова.

Варианты

Преимущество регулярного выражения, выполняющего проверку сразу для всех 27 стран, состоит в том, что в форму заказа достаточно будет добавить всего одно выражение. Можно было бы расширить форму заказа, используя 27 отдельных регулярных выражений. При этом сначала можно было бы определять страну, указанную в расчетном адресе, а затем выбирать регулярное выражение, соответствующее этой стране, из табл. 4.4.

Таблица 4.4. Регулярные выражения, совпадающие с номерами плательщиков НДС стран Евросоюза

Страна	Регулярное выражение, совпадающее с номером плательщика НДС
Австрия	‹^(AT)?U[0-9]{8}$›
Бельгия	‹^(BE)?0[0-9]{9}$›
Болгария	‹^(BG)?[0-9]{9,10}$›
Кипр	‹^(CY)?[0-9]{8}L$›
Чешская республика	‹^(CZ)?[0-9]{8,10}$›
Германия	‹^(DE)?[0-9]{9}$›
Дания	‹^(DK)?[0-9]{8}$›
Эстония	‹^(EE)?[0-9]{9}$›
Греция	‹^(EL|GR)?[0-9]{9}$›
Испания	‹^(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]$›
Финляндия	‹^(FI)?[0-9]{8}$›
Франция	‹^(FR)?[0-9A-Z]{2}[0-9]{9}$›
Великобритания	‹^(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})$›
Венгрия	‹^(HU)?[0-9]{8}$›
Ирландия	‹^(IE)?[0-9]S[0-9]{5}L$›
Италия	‹^(IT)?[0-9]{11}$›
Литва	‹^(LT)?([0-9]{9}|[0-9]{12})$›
Люксембург	‹^(LU)?[0-9]{8}$›
Латвия	‹^(LV)?[0-9]{11}$›

Страна	Регулярное выражение, совпадающее с номером плательщика НДС
Мальта	`‹^(MT)?[0-9]{8}$›`
Нидерланды	`‹^(NL)?[0-9]{9}B[0-9]{2}$›`
Польша	`‹^(PL)?[0-9]{10}$›`
Португалия	`‹^(PT)?[0-9]{9}$›`
Румыния	`‹^(RO)?[0-9]{2,10}$›`
Швеция	`‹^(SE)?[0-9]{12}$›`
Словения	`‹^(SI)?[0-9]{8}$›`
Словакия	`‹^(SK)?[0-9]{10}$›`

За основу для реализации проверки регистрационного номера выбранным регулярным выражением можно взять рецепт 3.6. Она позволит определить корректность регистрационного номера для страны, указанной клиентом.

Главное преимущество использования отдельных регулярных выражений состоит в том, что появляется возможность вставлять код страны в начало номера, не требуя этого от клиента. Если регулярное выражение соответствует введенному номеру, можно проверить содержимое первой сохраняющей группы. Как это сделать, описывается в рецепте 3.9. Если первая сохраняющая группа ничего не содержит, следовательно, клиент не ввел код страны. В этом случае можно добавить код страны перед номером, прежде чем сохранять его у себя в базе данных.

Греческие регистрационные номера могут содержать один из двух кодов страны. Код EL традиционно используется в греческих регистрационных номерах, а код GR является стандартизованным кодом страны для Греции.

См. также

Регулярное выражение просто проверяет, выглядит ли введенная последовательность как регистрационный номер плательщика НДС. Этого вполне достаточно, чтобы выявить непредумышленные ошибки. Регулярное выражение не проверяет, принадлежит ли введенный регистрационный номер компании, указанной в форме заказа. Проверить принадлежность регистрационного номера, если таковой вообще был кому-то присвоен, можно на странице *http://ec.europa.eu/taxation_customs/vies/vieshome.do*.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

5

Слова, строки и специальные символы

В этой главе содержатся рецепты, связанные с поиском и манипулированием текстом в различных контекстах. Одни рецепты демонстрируют, как реализовать расширенные функции поисковых механизмов, такие как поиск одного из нескольких слов или поиск слов, находящихся рядом друг с другом. Другие помогут вам отыскивать целые строки, содержащие определенные слова, удалять повторяющиеся слова или экранировать метасимволы регулярных выражений.

Основная цель этой главы состоит в том, чтобы продемонстрировать различные приемы и конструкции регулярных выражений в действии. Чтение этой главы сродни тренировке в применении большого числа особенностей регулярных выражений, что должно помочь наработать навыки применения регулярных выражений в решении стоящих перед вами задач. Во многих случаях мы ограничиваемся поиском простого текста, но предлагаемые шаблоны решений могут быть адаптированы вами под решение конкретных проблем.

5.1. Поиск определенного слова

Задача

Перед вами стоит простая задача – отыскать все вхождения слова `cat` без учета регистра символов. Выражение должно обнаруживать только те вхождения, которые являются отдельными словами. Оно не должно обнаруживать вхождения, являющиеся частью других слов, таких как `hellcat`, `application` или `Catwoman`.

Решение

Решить эту проблему помогут метасимволы границы слова:

```
\bcat\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Метасимволы границы слова, находящиеся с обоих концов регулярного выражения, гарантируют, что совпадение с текстом `cat` будет обнаруживаться, только если он появляется только как самостоятельное слово. Точнее, границы слова требуют, чтобы фрагмент `cat` отделялся от остального текста началом или концом строки, пробельными символами, знаками пунктуации или другими символами, не являющимися символами слова.

Механизмы регулярных выражений интерпретируют буквы, цифры и символ подчеркивания как символы слова. Впервые и достаточно подробно мы знакомились с границами слова в рецепте 2.6.

В диалектах JavaScript, PCRE и Ruby могут возникнуть проблемы с интернациональным текстом, так как в этих диалектах при определении границ слова в учет принимаются только символы кодировки ASCII. Другими словами, границы слова обнаруживаются только в позициях между совпадениями с выражениями `‹[^A-Za-z0-9_]|^›` и `‹[A-Za-z0-9_]›` или между совпадениями с выражениями `‹[A-Za-z0-9_]›` и `‹[^A-Za-z0-9_]|$›`. То же относится и к Python, если при создании регулярного выражения не был установлен флаг `UNICODE` или `U`. Эта особенность препятствует использованию метасимвола `‹\b›` для поиска «только целого слова» в тексте, содержащем символы из алфавитов, отличных от алфавита Latin. Например, в JavaScript, PCRE и Ruby регулярное выражение `‹\büber\b›` будет обнаруживать совпадение внутри слова `darüber` и не будет обнаруживать внутри текста `dar über`. В большинстве случаев такой результат представляет собой полную противоположность желаемому. Проблема обусловлена тем, что символ `ü` интерпретируется не как символ слова, поэтому метасимвол границы слова обнаруживает совпадение между символами `rü`. В свою очередь, между пробелом и символом `ü` граница слова не обнаруживается, потому что они образуют непрерывную последовательность символов, не являющихся символами слова.

Эту проблему можно решить, применив вместо метасимвола границы слова опережающую и ретроспективную проверки (*проверки соседних символов* – см. рецепт 2.6). Подобно метасимволу границы слова проверка соседних символов обеспечивает совпадение нулевой длины с некоторой позицией. В диалектах PCRE (когда эта библиотека скомпилирована с поддержкой UTF-8) и Ruby 1.9 имеется возможность имитировать поддержку границы слова в кодировке Юникод, например `‹(?<=[^\p{L}\p{M}]|^)cat(?=[^\p{L}\p{M}]|$)›`. В этом регулярном выражении использованы метасимволы категорий Юникода Letter и Mark (`‹\p{L}›` и `‹\p{M}›`), которые рассматривались в рецепте 2.7. Если потребуется,

чтобы проверка соседних символов интерпретировала десятичные цифры и знак-соединитель (символ подчеркивания и сходные с ним) как символы слова (подобно метасимволу ‹\b›), замените обе конструкции ‹[^\p{L}\p{M}]› символьным классом ‹[^\p{L}\p{M}\p{Nd}\p{Pc}]›.

В JavaScript и Ruby 1.8 не поддерживаются ни ретроспективная проверка, ни категории Юникода. Обойти отсутствие поддержки ретроспективной проверки в этих диалектах можно сопоставлением с символом, не являющимся символом слова, перед каждым совпадением, а затем либо удалением его из каждого совпадения в программном коде, либо возвращением его обратно в строку при выполнении операции замещения (примеры использования фрагментов совпадений в замещающем тексте приводятся в рецепте 3.15). Кроме того, отсутствие поддержки категорий Юникода (вместе с тем фактом, что в обоих языках программирования метасимволы ‹\w› и ‹\W› поддерживают только символы ASCII) означает, что придется обходиться менее универсальным решением. Кодовые пункты из категорий Letter и Mark рассеяны по всему набору символов Юникода, поэтому потребовались бы тысячи символов, чтобы имитировать конструкцию ‹[^\p{L}\p{M}]› с помощью экранированных последовательностей Юникода и символьных классов. Неплохим компромиссом мог бы стать символьный класс ‹[^A-Za-z\xAA\xB5\xBA\xC0-\xD6\xD8-\xF6\xF8-\xFF]›, которому соответствуют все буквы (кроме букв Юникода) в восьмибитовом адресном пространстве (то есть первые 256 кодов Юникода со значениями от 0x00 до 0xFF). Список соответствующих символов приводится на рис. 5.1. Этот символьный класс будет исключать множество (или в неинвертированной форме совпадать с множеством) наиболее часто используемых символов с диакритическими знаками.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
:																
4		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z					
6		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z					
:																
A											ª					
B						µ					º					
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö		Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö		ø	ù	ú	û	ü	ý	þ	ÿ

***Рис. 5.1.** Алфавитные символы Юникода в восьмибитовом адресном пространстве*

Ниже демонстрируется, как можно заменить все вхождения слова «cat» словом «dog» в JavaScript. Этот пример корректно опознает все наиболее типичные акцентированные символы, поэтому слово écat останется без изменений. Для этого потребовалось сконструировать собственный символьный класс и отказаться от использования встроенных метасимволов <\b> и <\w>:

```
// 8-битные алфавитные символы
var pL = "A-Za-z\xAA\xB5\xBA\xC0-\xD6\xD8-\xF6\xF8-\xFF",
    pattern = "([^{L}]|^)cat([^{L}]|$)".replace(/{L}/g, pL),
    regex = new RegExp(pattern, "gi");

// заменить cat на dog и вернуть в строку
// любые совпавшие дополнительные символы
subject = subject.replace(regex, "$1dog$2");
```

Примечательно, что для вставки специальных символов в строковый литерал в JavaScript используется форма записи \x*HH* (где *HH* – это двузначное шестнадцатеричное число). Следовательно, переменная pL, которая передается в регулярное выражение, будет содержать литеральные версии символов. Если необходимо, чтобы в регулярное выражение передавались сами метапоследовательности \x*HH*, их необходимо экранировать в строковом литерале с помощью символа обратного слэша (например, "\\x*HH*") . Однако в данном случае это не имеет большого значения и не окажет влияния на совпадение с регулярным выражением.

См. также

В этой главе вы встретитесь с множеством рецептов, реализующих сопоставление с отдельными словами. В рецепте 5.2 объясняется, как отыскать любое из множества слов. В рецепте 5.3 описывается, как искать похожие слова. В рецепте 5.4 рассказывается, как найти все слова, кроме определенного. В рецепте 5.10 демонстрируется, как найти совпадение с полными строками, содержащими определенное слово.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.7 рассказывается о сопоставлении с символами Юникода. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

5.2. Поиск любого слова из множества

Задача

Необходимо отыскать одно из слов из списка, не выполняя несколько операций поиска в испытуемом тексте.

Решение

С использованием конструкции выбора

Самое простое решение заключается в создании конструкции выбора из требуемых слов:

```
\b(?:one|two|three)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Более сложные примеры поиска похожих слов рассматриваются в рецепте 5.3.

Пример решения на языке JavaScript

```
var subject = "One times two plus one equals three.";

// Решение 1:

var regex = /\b(?:one|two|three)\b/gi;

subject.match(regex);
// вернет массив с четырьмя совпадениями: ["One","two","one","three"]

// Решение 2 (повторно используемое):

// Эта функция делает то же самое, но принимает искомые слова
// в виде массива. Любые метасимволы регулярных выражений,
// присутствующие внутри искомых слов, перед поиском
// экранируются символами обратного слэша.
function matchWords(subject, words) {
    var regexMetachars = /[(){[*+?.\\^$|]/g;

    for (var i = 0; i < words.length; i++) {
        words[i] = words[i].replace(regexMetachars, "\\$&");
    }

    var regex = new RegExp("\\b(?:" + words.join("|") + ")\\b", "gi");

    return subject.match(regex) || [];
}

matchWords(subject, ["one","two","three"]);
// вернет массив с четырьмя совпадениями: ["One","two","one","three"]
```

Обсуждение

С использованием конструкции выбора

Это регулярное выражение состоит из трех частей: границы слова с обоих концов, несохраняющая группа и список слов (отделяются друг от друга оператором выбора ‹|›). Границы слова гарантируют, что регуляр-

ное выражение не совпадет с частью более длинного слова. Несохраняющая группировка ограничивает область действия операторов выбора; в противном случае для достижения того же эффекта регулярное выражение пришлось бы записать как ‹\bone\b|\btwo\b|\bthree\b›. Каждое из слов должно просто совпасть литерально.

Так как слова с обеих сторон окружены границами слова, они могут появляться в любом порядке. Однако если бы в выражении отсутствовали границы слова, то важно было бы первыми указать более длинные слова, в противном случае выражение ‹awe|awesome› никогда не смогло бы найти совпадение со словом «awesome», так как оно всегда обнаруживало бы совпадение с «awe» в начале слова.

Так как механизм регулярных выражений пытается найти совпадение с каждым словом из списка, просматривая его слева направо, можно добиться некоторого повышения производительности, если в начало списка поместить слова, которые могут встречаться в испытуемом тексте с большей долей вероятности.

Обратите внимание, что это регулярное выражение предназначено лишь для того, чтобы в общих чертах продемонстрировать возможность совпадения с одним словом из списка. Поскольку в данном примере оба слова ‹two› и ‹three› начинаются с одного и того же символа, есть возможность помочь механизму регулярных выражений, переписав регулярное выражение как ‹\b(?:one|t(?:wo|hree))\b›. Однако на практике не стоит доходить до крайности в подобной тонкой ручной оптимизации. Большинство механизмов регулярных выражений применяют подобные оптимизации автоматически, по крайней мере в наиболее простых случаях. В рецепте 5.3 приводятся дополнительные примеры, демонстрирующие, как эффективнее выполнять поиск одного слова из списка похожих слов.

Пример решения на языке JavaScript

Пример на языке JavaScript сопоставляет тот же список слов двумя разными способами. В первом случае просто создается регулярное выражение и с помощью метода `match()`, имеющегося у строк в языке JavaScript, выполняется поиск в испытуемом тексте. При вызове методу `match()` передается регулярное выражение, использующее флаг `/g` («global» – глобальный). Он возвращает массив всех совпадений, обнаруженных в строке, или значение `null`, если не найдено ни одного совпадения.

Во втором случае используется функция `matchWords()`, принимающая испытуемую строку, внутри которой требуется выполнить поиск, и массив искомых слов. Эта функция сначала экранирует любые метасимволы регулярных выражений, которые могут присутствовать в искомых словах (рецепт 2.1), и компонует из списка слов новое регулярное выражение, используемое для поиска в строке. Затем это регулярное выражение используется для поиска в строке сразу всех искомых слов вместо перебора их в цикле. Функция возвращает массив обнаруженных совпаде-

ний или пустой массив, если сгенерированным регулярным выражением не было найдено ни одного совпадения. Благодаря применению флага нечувствительности к регистру символов (/i) искомые слова могут состоять из символов верхнего и нижнего регистра в любой комбинации.

См. также

В этой главе вы встретитесь с множеством рецептов, реализующих сопоставление с отдельными словами. В рецепте 5.1 объясняется, как отыскать определенное слово. В рецепте 5.3 описывается, как искать похожие слова. В рецепте 5.4 рассказывается, как найти все слова, кроме определенного.

В рецепте 4.11 показано решение, проверяющее утвердительные ответы, которое также выполняет поиск совпадения с любым из множества слов.

Некоторые языки программирования имеют встроенные функции, выполняющие экранирование метасимволов регулярных выражений, как показано в рецепте 5.14.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке.

5.3. Поиск похожих слов

Задача

В данном случае необходимо решить несколько задач:

- Отыскать все вхождения обоих слов, `color` и `colour`, в строке.
- Отыскать любое из трех слов, оканчивающееся на «at»: `bat`, `cat` или `rat`.
- Отыскать любое слово, оканчивающееся на `phobia`.
- Отыскать наиболее распространенные варианты записи имени «Steven»: `Steve`, `Steven` и `Stephen`.
- Отыскать совпадение с любой формой термина «regular expression».

Решение

Ниже приводятся регулярные выражения, решающие поставленные задачи в порядке их следования. Во всех решениях используется режим нечувствительности к регистру символов.

Color или colour

```
\bcolou?r\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Bat, cat или rat

```
\b[bcr]at\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Слова, заканчивающиеся на «phobia»

```
\b\w*phobia\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Steve, Steven или Stephen

```
\bSte(?:ven?|phen)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Варианты написания термина «regular expression»

```
\breg(?:ular*expressions?|ex(?:ps?|e[sn])?)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Использование границ слова для обеспечения совпадения с целыми словами

Для обеспечения совпадения с целыми словами во всех пяти регулярных выражениях используются метасимволы границы слова (‹\b›) . Во всех шаблонах используются разные способы сопоставления с вариантами слов, которым они соответствуют.

Рассмотрим каждое из решений подробнее.

Color или Colour

Это регулярное выражение будет совпадать со словом color или colour, но не будет совпадать со словом colorblind. В нем используется квантификатор ‹?›, чтобы сделать необязательным совпадение с предшествующим ему символом u. Квантификаторы, такие как ‹?›, действуют не как шаблонные символы, знакомые многим. Они непосредственно связаны с предшествующими им элементами, которые могут представлять собой либо одиночный элемент (как литерал символа u в данном случае), либо группу элементов, заключенных в круглые скобки. Квантификатор ‹?› повторяет предшествующий ему элемент ноль или один раз. Механизм регулярных выражений сначала пытается сопоставить элемент, с которым связан квантификатор, и если эта попытка оказывается неудачной, механизм продолжает движение вперед без сопоставления элемента. Любой квантификатор, допускающий нулевое число повторе-

ний, фактически делает необязательным совпадение с предшествующим ему элементом, и это как раз то, что требуется в данном случае.

Bat, cat или rat

В этом регулярном выражении для совпадения с символом `b`, `c` или `r` используется символьный класс, за которым следуют литералы символов `at`. То же самое можно было бы сделать с помощью выражений `‹\b(?:b|c|r)at\b›`, `‹\b(?:bat|cat|rat)\b›` или `‹\bbat\b|\bcat\b|\brat\b›`. Однако всякий раз когда выбор между возможными совпадениями заключается в выборе единственного символа из списка, лучше использовать символьные классы. Мало того что символьные классы обеспечивают более компактную и удобочитаемую форму записи (благодаря отсутствию операторов выбора и возможности использовать диапазоны, такие как A–Z), большинство механизмов регулярных выражений также обеспечивают превосходную оптимизацию обработки символьных классов. Операторы выбора требуют от механизма регулярных выражений использовать дорогостоящий, в смысле вычислительных ресурсов, алгоритм возвратов, тогда как для сопоставления с символьными классами задействуется более простой метод поиска.

Тем не менее несколько слов предостережения. Символьные классы относятся к элементам регулярных выражений, которые наиболее часто применяются неверно. Возможно, проблема в недостаточно подробной документации или в недостаточно внимательном ознакомлении с нею. Как бы то ни было, не повторяйте ошибок, характерных для начинающих. Символьные классы способны обеспечить совпадение лишь с одним из символов, перечисленных в них, и только это.

Ниже приводятся два наиболее типичных примера неправильного использования символьных классов:

Помещение слов в символьные классы

> Вне всяких сомнений, выражение `‹[cat]{3}›` будет обнаруживать совпадение со словом `cat`, но оно также будет совпадать со словами `act`, `ttt` и любыми другими комбинациями из перечисленных символов. То же относится и к инвертированным классам, таким как `‹[^cat]›`, которому соответствует любой символ, за исключением `c`, `a` и `t`.

Попытка использовать оператор выбора в символьном классе

> Символьный класс по определению обеспечивает выбор между символами, перечисленными в нем. Конструкции `‹[a|b|c]›` соответствует один символ из множества «abc|». Что, вероятно, совсем не то, что вы подразумевали. Но даже если это не так, данный символьный класс содержит лишние символы вертикальной черты.

Все подробности, которые могут пригодиться для правильного и эффективного использования символьных классов, приводятся в рецепте 2.3.

Слова, заканчивающиеся на «phobia»

В этом шаблоне используются особенности, применявшиеся в двух предыдущих регулярных выражениях. Как и в выражении из подраздела «Bat, cat или rat», здесь применяется символьный класс (сокращение ‹\w›), совпадающий с любым символом слова. За ним следует квантификатор ‹*›, пытающийся найти ноль или более совпадений с предыдущим символьным классом, подобно квантификатору ‹?› в регулярном выражении из раздела «Color или colour».

Этому регулярному выражению соответствуют такие слова, как arachnophobia и hippopotomonstrosesquipedaliophobia. Так как квантификатор ‹*› допускает нулевое число повторений, ему также будет соответствовать само слово phobia. Если необходимо, чтобы перед окончанием «phobia» присутствовал хотя бы один символ, следует заменить квантификатор ‹*› на ‹+›.

Steve, Steven или Stephen

В дополнение к другим средствам поиска альтернатив в данное регулярное выражение был добавлен оператор выбора. Несохраняющая группировка, записываемая как ‹(?:···)›, ограничивает область действия оператора выбора ‹|›. Квантификатор ‹?›, применяемый к первой альтернативе внутри группы, обеспечивает необязательность предшествующего ему символа ‹n›. Это увеличивает эффективность (и краткость) выражения по сравнению с эквивалентным выражением ‹\bSte(?:ve|ven|phen)\b›. Те же соображения эффективности заставляют поместить строковый литерал ‹Ste› в начало регулярного выражения, а не повторять его трижды, как в выражениях ‹\b(?:Steve|Steven|Stephen)\b› или ‹\bSteve\b|\bSteven\b|\bStephen\b›. Некоторые механизмы возвратов недостаточно интеллектуальны, чтобы заметить, что любой текст, соответствующий этим двум последним выражениям, должен начинаться с последовательности символов Ste. По мере продвижения механизма по испытуемой строке в поисках совпадения он сначала попытается отыскать границу слова и затем убедиться, что следующий символ – это символ S. Если совпадение не найдено, механизм вынужден будет опробовать все возможные альтернативы, присутствующие в регулярном выражении, прежде чем он сможет переместиться к следующей позиции в строке и повторить попытку. Если человек легко может заметить, что это будет напрасной тратой сил (так как все альтернативы в регулярном выражении начинаются с последовательности Ste), то механизм даже не подозревает об этом. Если же выражение записано как ‹\bSte(?:ven?|phen)\b›, механизм тут же сообразит, что он не сможет отыскать соответствие в строке, которая не начинается с указанных символов.

Более подробно о работе механизма возвратов рассказывается в рецепте 2.13.

Варианты написания термина «regular expression»

Последний пример в этом рецепте соединяет в себе конструкцию выбора, символьные классы и квантификаторы, чтобы отыскать совпадение со всеми типичными вариантами записи термина «regular expression». Поскольку на первый взгляд это регулярное выражение может показаться немного сложным, разобьем его на составляющие и рассмотрим их по отдельности.

Регулярное выражение, которое приводится ниже, записано в режиме свободного форматирования, который не поддерживается диалектом JavaScript. Поскольку в режиме свободного форматирования пробелы игнорируются, литералы пробелов были экранированы символами обратного слэша:

```
\b              # Проверка совпадения с границей слова.
reg             # Соответствует "reg".
(?:             # Группировка, но несохраняющая...
  ular\         #   Соответствует "ular ".
  expressions?  #   Соответствует "expression" или "expressions".
 |              #  Или:
  ex            #   Соответствует "ex".
  (?:           #   Группировка, но несохраняющая...
    ps?         #     Соответствует "p" или "ps".
   |            #    Или:
    e[sn]       #     Соответствует "es" или "en".
  )?            #   Конец несохраняющей группы. Сделать ее необязательной.
)               # Конец несохраняющей группы.
\b              # Проверка совпадения с границей слова.
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Данный шаблон совпадет с любой из следующих семи строк, содержащих любые комбинации букв верхнего и нижнего регистра:

- regular expressions
- regular expression
- regexps
- regexp
- regexes
- regexen
- regex

См. также

В рецепте 5.1 объясняется, как отыскать определенное слово. В рецепте 5.2 объясняется, как отыскать любое из множества слов. В рецепте 5.4 рассказывается, как найти все слова, кроме определенного.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

5.4. Поиск любых слов, за исключением указанного

Задача

Необходимо с помощью регулярного выражения отыскать совпадения с любыми полными словами, *за исключением* слова `cat`. Слова `Catwoman`, `vindicate` и другие подобные слова, которые просто содержат в себе символы «cat», должны совпадать, а слово `cat` – нет.

Решение

Исключить совпадения с нежелательными словами можно с помощью негативной опережающей проверки, которая является ключевым элементом следующего регулярного выражения:

```
\b(?!cat\b)\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Несмотря на то что с помощью инвертированного символьного класса (записывается как ‹[^···]›) можно легко исключить возможность совпадения с определенными символами, тем не менее нельзя просто записать ‹[^cat]› и надеяться, что будут исключены совпадения со словом `cat`. ‹[^cat]› – это допустимое регулярное выражение, но ему соответствует любой символ, за исключением символов `c`, `a` и `t`. Поэтому, хотя выражение ‹\b[^cat]+\b› могло бы помочь исключить совпадения со словом `cat`, оно точно так же исключило бы совпадения со словом `time`, потому что в этом слове содержится запрещенный символ `t`. Регулярное выражение ‹\b[^c][^a][^t]\w*› ничуть не лучше, потому что оно будет отвергать любые слова, в которых первым следует символ `c`, вторым – `a` или третьим – `t`. Кроме того, это регулярное выражение не накладывает ограничений на первые три символа слова – оно лишь совпадает со словами, содержащими, по крайней мере, три символа, поскольку ни один из инвертированных символьных классов не является необязательным.

Учитывая все вышесказанное, рассмотрим, как регулярное выражение, представленное в начале этого рецепта, позволяет решить предложенную задачу:

```
\b    # Проверка соответствия границе слова.
(?!   # Проверка, что регулярное выражение, следующее ниже, не найдет
      # совпадения, начиная с этой позиции...
  cat #   Соответствие "cat".
  \b  #   Проверка соответствия границе слова.
)     # Конец негативной опережающей проверки.
\w+   # Соответствует одному или более символу слова.
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Ключевым элементом этого шаблона является негативная опережающая проверка ‹(?!···)›. Она отклоняет возможность совпадения с последовательностью символов `cat`, за которой следует граница слова, и не запрещает регулярному выражению совпадать с этими же символами, если они не следуют именно в таком порядке или когда они составляют часть более длинного или более короткого слова. В конце регулярного выражения отсутствует проверка совпадения с границей слова, потому что ее присутствие не повлияет на возможность совпадения с регулярным выражением. Квантификатор ‹+› в подвыражении ‹\w+› будет повторять попытки найти совпадение с символом слова максимально возможное число раз, то есть он всегда будет обнаруживать совпадение до ближайшей границы слова.

Если применить это регулярное выражение к испытуемому тексту `categorically match any word except cat`, оно обнаружит пять совпадений: categorically, match, any, word и except.

Варианты

Поиск слов, не содержащих других слов

Если вместо того чтобы пытаться отыскать совпадения с любыми словами, кроме слова `cat`, необходимо отыскать любые слова, которые *не содержат* в себе слово `cat`, требуется немного иной подход:

```
\b(?:(?!cat)\w)+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В предыдущем разделе этого рецепта метасимвол границы слова в начале регулярного выражения обеспечивал удобную привязку, которая позволила просто поместить негативную опережающую проверку в начало слова. Решение, используемое здесь, не столь эффективно; тем не менее эта конструкция часто используется, чтобы обеспечить возможность совпадения с любым словом, за исключением определенного слова или шаблона. Это достигается за счет повторения группы, содержащей негативную опережающую проверку и единственный метасимвол, обозначающий символ слова. Перед сопоставлением с каждым симво-

лом механизм регулярных выражений проверяет отсутствие совпадения со словом cat, начиная с текущей позиции.

В отличие от предыдущего регулярного выражения, в данном выражении требуется присутствие завершающего метасимвола границы слова. В противном случае оно могло бы совпасть с первой частью слова перед вхождением в него слова cat.

Если применить это регулярное выражение к испытуемому тексту categorically match any word except cat, оно обнаружит четыре совпадения: match, any, word и except.

См. также

В рецепте 5.1 объясняется, как отыскать определенное слово. В рецепте 5.5 объясняется, как отыскать любое слово, за которым не следует указанное слово. В рецепте 5.6 описывается, как отыскать любое слово, которому не предшествует указанное слово. В рецепте 5.11 демонстрируется, как найти совпадение с полными строками, не содержащими определенное слово.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

5.5. Поиск любого слова, за которым не следует указанное слово

Задача

Необходимо отыскать совпадение с любым словом, непосредственно за которым не следует слово cat, игнорируя присутствующие между ними любые пробельные символы, знаки пунктуации и другие символы, не являющиеся символами слова.

Решение

Секретным ингредиентом этого рецепта является негативная опережающая проверка:

```
\b\w+\b(?!\W+cat\b)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Как и во многих других рецептах этой главы, совпадение с целым словом обеспечивается совместной работой метасимволов границы слова (‹\b›) и символа слова (‹\w›). Более подробное описание этих особенностей можно найти в рецепте 2.6.

Вторая часть этого выражения заключена в конструкцию ‹(?!…)›, которая представляет собой негативную опережающую проверку. Опережающая проверка предписывает механизму регулярных выражений временно пройти вперед по строке, чтобы проверить возможность совпадения текста, находящегося непосредственно за текущей позицией, с шаблоном, находящимся внутри опережающей проверки. При этом любые символы, совпавшие с выражением внутри опережающей проверки, не поглощаются – проверка просто убеждается, что совпадение возможно. Поскольку здесь используется *негативная* опережающая проверка, результат проверки инвертируется. Другими словами, если шаблон внутри опережающей проверки может обнаружить совпадение непосредственно за текущей позицией, попытка сопоставления считается неудачной и механизм регулярных выражений продвигается на одну позицию вперед, чтобы предпринять новую попытку сопоставления со всем регулярным выражением, начиная со следующего символа в испытуемой строке. Более подробное обсуждение опережающей (и ретроспективной) проверки можно найти в рецепте 2.16.

Что касается шаблона внутри опережающей проверки: конструкции ‹\W+› соответствует один или более символов, не являющихся символами слова, таких как знаки препинания и пробел, которые находятся перед словом ‹cat›, а завершающий метасимвол границы слова гарантирует, что совпадение будет обнаружено только со словами, за которыми не следует последовательность символов `cat` в виде самостоятельного слова, но допускается, что следующее слово начинается с символов `cat`.

Следует заметить, что это регулярное выражение будет совпадать даже со словом `cat` при условии, что за ним не следует второе слово `cat`. Если потребуется избежать совпадения со словом `cat`, следует объединить это регулярное выражение с регулярным выражением из рецепта 5.4, что в результате даст выражение ‹\b(?!cat\b)\w+\b(?!\W+cat\b)›.

Варианты

Если необходимо обеспечить совпадение только со словами, *за* которыми следует слово `cat` (не включая в совпадение само слово `cat` и предшествующие ему символы, не являющиеся символами слова), замените негативную опережающую проверку на позитивную и радуйтесь жизни:

```
\b\w+\b(?=\W+cat\b)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

В рецепте 5.4 рассказывается, как найти все слова, кроме определенного. В рецепте 5.6 описывается, как отыскать любое слово, которому не предшествует указанное слово.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

5.6. Поиск любого слова, которому не предшествует определенное слово

Задача

Необходимо отыскать любое слово, которое не следует непосредственно за словом `cat`, игнорируя присутствующие между ними любые символы, не являющиеся символами слова.

Решение

Взгляд назад

Ретроспективная проверка позволяет проверить наличие некоторого текста непосредственно перед текущей позицией. Она предписывает механизму регулярных выражений временно вернуться назад по строке, чтобы проверить возможность совпадения текста, оканчивающегося в позиции, где находится ретроспективная проверка. Более подробное обсуждение ретроспективной проверки можно найти в рецепте 2.16.

Следующие регулярные выражения используют негативную ретроспективную проверку ‹(?<!…)›. К сожалению, диалекты регулярных выражений, рассматриваемые в этой книге, отличаются допустимыми шаблонами, которые можно помещать внутрь ретроспективной проверки. В результате получившиеся решения работают немного по-разному. Дополнительные подробности можно найти в разделе «Обсуждение» ниже в этом рецепте.

Слова, не предшествующие слову «cat»

Любое число разделяющих символов, не являющихся символами слова:

```
(?<!\bcat\W+)\b\w+
```

Параметры: нечувствительность к регистру символов
Диалект: .NET

Ограниченное число разделяющих символов, не являющихся символами слова:

```
(?<!\bcat\W{1,9})\b\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java

Единственный разделяющий символ, не являющийся символом слова:

```
(?<!\bcat\W)\b\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python

```
(?<!\Wcat\W)(?<!^cat\W)\b\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Имитация ретроспективной проверки

Диалекты JavaScript и Ruby 1.8 вообще не поддерживают ретроспективную проверку, хотя и предоставляют возможность опережающей проверки. Однако благодаря тому, что в данной задаче ретроспективная проверка находится в самом начале регулярного выражения, имеется отличная возможность имитировать ретроспективную проверку, разбив регулярное выражение на две части, как показано в следующем примере на JavaScript:

```
var subject = "My cat is fluffy.",
    mainRegex = /\b\w+/g,
    lookbehind = /\bcat\W+$/i,
    lookbehindType = false, // false - негативная ретроспективная проверка
                            // true - позитивная ретроспективная проверка
    matches = [],
    match,
    leftContext;

while (match = mainRegex.exec(subject)) {
    leftContext = subject.substring(0, match.index);

    if (lookbehindType == lookbehind.test(leftContext)) {
        matches.push(match[0]);
    } else {
        mainRegex.lastIndex = match.index + 1;
    }
}

// совпадения: ["My", "cat", "fluffy"]
```

Обсуждение

Фиксированная, конечная и бесконечная длина совпадения с ретроспективной проверкой

В первом регулярном выражении используется негативная ретроспективная проверка `(?<!\bcat\W+)`. Так как квантификатор `+`, используемый в ретроспективной проверке, не имеет верхнего предела числа совпавших символов, эта версия будет работать только в диалекте регулярных выражений .NET. Все остальные диалекты, рассматриваемые в этой книге, требуют, чтобы совпадение с шаблоном в ретроспективной проверке имело фиксированную или конечную длину.

Во втором регулярном выражении квантификатор `+` в ретроспективной проверке был заменен квантификатором `{1,9}`. Поэтому такое регулярное выражение может использоваться в диалектах .NET и Java, которые поддерживают возможность совпадения с ретроспективной проверкой переменной длины, если заранее известен верхний предел числа совпадающих символов. Для примера я ограничил максимальное число символов, не являющихся символами слова, разделяющих отдельные слова, девятью. Это допускает возможность появления нескольких пробелов и знаков пунктуации между словами. Если только испытуемый текст не представляет собой нечто необычное, это регулярное выражение будет работать точно так же, как и решение, применимое только в диалекте .NET. Однако даже в диалекте .NET указание верхнего предела повторений для любых квантификаторов внутри ретроспективной проверки сделает регулярное выражение более безопасным, хотя бы потому, что в данном случае уменьшается число непредвиденных возвратов, которые могут возникнуть внутри ретроспективной проверки.

В третьем регулярном выражении вообще отсутствует квантификатор после метасимвола `\W` внутри ретроспективной проверки. Для ретроспективной проверки это дает возможность просматривать строку фиксированной длины, что обеспечило поддержку диалектов PCRE, Perl и Python. Но цена оказалась слишком высока, и теперь регулярное выражение просто не будет совпадать со словами, которым предшествует слово «cat», отделяемое от анализируемого слова единственным разделяющим символом. Регулярное выражение будет правильно совпадать только со словом `cat` в словосочетании `cat fluff`, но оно ошибочно будет совпадать с обоими словами, `cat` и `fluff`, в строке `cat, fluff`.

Так как Ruby 1.9 не позволяет использовать границы слов `\b` в ретроспективных проверках, в четвертом регулярном выражении используются две отдельные ретроспективные проверки. Первая отвергает слово `cat`, если оно является предшествующим и ему самому предшествует символ, не являющийся символом слова, такой как пробел или знак пунктуации. Вторая использует якорный метасимвол `^`, чтобы предотвратить совпадение с предшествующим словом `cat`, если оно находится в начале строки.

Имитация ретроспективной проверки

Диалект JavaScript не поддерживает ретроспективную проверку, но в примере программного кода на JavaScript показано, как можно имитировать обратную проверку, которая должна выполняться в начале регулярного выражения. В этом примере нет ограничений на длину совпадения с (имитируемой) ретроспективной проверкой.

Для начала мы разбили регулярное выражение `‹(?<!\bcat\W+)\b\w+›` из первого решения на две части: выражение внутри ретроспективной проверки (`‹\bcat\W+›`) и выражение, следующее за ней (`‹\b\w+›`). Затем добавили метасимвол `‹$›` в конец регулярного выражения, выполняющего ретроспективную проверку. Если регулярное выражение `lookbehind` потребуется применять в режиме «символ крышки и знак доллара соответствуют границам строк» (модификатор `/m`), то вместо `‹$›` следует вставить конструкцию `‹$(?!\s)›`, чтобы гарантировать совпадение только с концом испытуемого текста. Переменная `lookbehindType` указывает, является имитируемая ретроспективная проверка позитивной или негативной: значение `true` обозначает позитивную, а `false` – негативную ретроспективную проверку.

После инициализации всех переменных с помощью методов `mainRegex()` и `exec()` выполняется обход испытуемого текста (описание этого процесса приводится в рецепте 3.11). После того как будет найдено соответствие, фрагмент испытуемого текста, предшествующий совпадению, копируется в новую строковую переменную (`leftContext`) и производится поиск совпадений с регулярным выражением `lookbehind` в этой строке. Так как в конец регулярного выражения `lookbehind` был добавлен якорь, это гарантирует, что второе совпадение будет находиться непосредственно перед совпадением, найденным объектом `mainRegex`, то есть в конце `leftContext`. Сравнивая результат выполнения ретроспективной проверки с переменной `lookbehindType`, можно определить, соответствует ли установленный критерий успешному совпадению.

В заключение выполняется одно из двух возможных действий. Если совпадение признано успешным, фрагмент, найденный объектом `mainRegex`, добавляется в массив `matches`. Если нет, то позиция, с которой будет продолжен поиск совпадения (для этого используется свойство `mainRegex.lastIndex`), смещается на один символ вперед от начальной позиции последнего совпадения объекта `mainRegex`, чтобы метод `exec()` не начал следующую итерацию от конца текущего совпадения.

Все! Мы закончили!

Этот сложный трюк задействует свойство `lastIndex`, значение которого динамически обновляется для регулярных выражений, использующих флаг `/g` («global» – глобальный). Обычно обновление и сброс значения свойства `lastIndex` происходят автоматически. Здесь мы воспользовались им, чтобы взять под свой контроль порядок прохождения испытуемого текста регулярным выражением, перемещая позицию поиска

совпадения вперед или назад по мере необходимости. Эта хитрость помогает имитировать только ретроспективную проверку, находящуюся в начале регулярного выражения. С некоторыми изменениями этот программный код мог бы также использоваться для имитации ретроспективной проверки, находящейся в самом конце регулярного выражения. Однако он не может обеспечить полную поддержку ретроспективной проверки. Из-за необходимости организовать взаимодействие между механизмом ретроспективной проверки и механизмом возвратов такой подход не может использоваться для точной имитации поведения ретроспективной проверки, находящейся в середине регулярного выражения.

Варианты

Если необходимо отыскать совпадение только со словами, которым предшествует слово `cat` (исключив из совпадения само слово `cat` и разделяющие их символы, не являющиеся символами слова), следует заменить негативную ретроспективную проверку позитивной:

Любое число разделяющих символов, не являющихся символами слова:

```
(?<=\bcat\W+)\w+
```

Параметры: нечувствительность к регистру символов
Диалект: .NET

Ограниченное число разделяющих символов, не являющихся символами слова:

```
(?<=\bcat\W{1,9})\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java

Единственный разделяющий символ, не являющийся символом слова:

```
(?<=\bcat\W)\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python

```
(?:(?<=\Wcat\W)|(?<=^cat\W))\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Эти адаптированные версии больше не содержат метасимвол границы слова ‹`\b`› перед ‹`\w+`› в конце регулярных выражений, потому что позитивная ретроспективная проверка уже гарантирует наличие перед совпадением какого-либо символа, не являющегося символом слова. Последнее регулярное выражение (добавляющее поддержку диалекта Ruby 1.9) обертывает две позитивные ретроспективные проверки конструкцией ‹`(?:…|…)`›, потому что в данной позиции может совпадать только одна ретроспективная проверка.

PCRE 7.2 и Perl 5.10 поддерживают необычный оператор ‹\K› (keep – запомнить), переустанавливающий начальную позицию для части совпадения, возвращаемого в результате сопоставления (подробнее об этом метасимволе рассказывается в разделе «Альтернатива ретроспективной проверке» в рецепте 2.16). Мы можем использовать этот метасимвол для создания достаточно близкой имитации позитивной ретроспективной проверки, не имеющей ограничений на длину совпадения, как показано ниже:

```
\bcat\W+\K\w+
```

Параметры: нечувствительность к регистру символов
Диалекты: PCRE 7.2, Perl 5.10

Между этим регулярным выражением и версией, поддерживающей только диалект .NET и допускающей произвольное количество разделяющих символов, не являющихся символами слова, существует одно тонкое, но важное отличие. В отличие от ретроспективной проверки, текст, совпавший с частью выражения слева от ‹\K›, поглощается при сопоставлении, даже при том что он не включается в общий результат. Эту разницу можно заметить, сравнив результаты сопоставления испытуемой строки `cat cat cat cat` с регулярными выражениями, содержащими ‹\K› и позитивную ретроспективную проверку. В Perl и PHP, если заменить все совпадения с ‹(?<=\bcat\W)\w+› на «dog», в результате получится `cat dog dog dog`, так как только первому слову не предшествует слово `cat`. Если воспользоваться регулярным выражением ‹\bcat\W+\K\w+›, чтобы выполнить ту же замену, в результате получится строка `cat dog cat dog`. После обнаружения совпадения с первой парой слов `cat cat` (и замены на `cat dog`), следующая попытка сопоставления не сможет заглянуть влево от своей начальной позиции, как это делает ретроспективная проверка. Регулярное выражение найдет совпадение со второй парой слов `cat cat`, которую опять заменит на `cat dog`.

См. также

Рецепт 5.4, в котором рассказывается, как найти все слова, кроме определенного. В рецепте 5.5 объясняется, как отыскать любое слово, за которым не следует указанное слово.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках. В этом же рецепте в разделе «Альтернатива ретроспективной проверке» описывается метасимвол ‹\K›.

5.7. Поиск близко расположенных слов

Задача

Необходимо с помощью регулярного выражения имитировать NEAR-поиск слов, стоящих поблизости. Для тех, кто не знаком с этим термином, поясню: в некоторых средствах поиска, использующих логические операторы, такие как NOT или OR, имеется также специальный оператор NEAR. При попытке выполнить поиск по строке «word1 NEAR word2» будут найдены слова word1 и word2, следующие в любом порядке, при условии, что они расположены друг от друга в пределах некоторого расстояния.

Решение

Если требуется просто отыскать два разных слова, можно объединить два регулярных выражения, одно из которых совпадает со словом word1, находящимся перед word2, и второе, совпадающее с искомыми словами, следующими в обратном порядке. Следующее регулярное выражение позволяет отыскивать пары слов, которые могут разделять до пяти других слов:

```
\b(?:word1\W+(?:\w+\W+){0,5}?word2|word2\W+(?:\w+\W+){0,5}?word1)\b
```

Параметры: нечувствительность к регистру символов

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b(?:
  word1                  # первое искомое слово
  \W+ (?:\w+\W+){0,5}?   # до пяти слов
  word2                  # второе искомое слово
 |                       #   или шаблон из тех же слов в обратном порядке...
  word2                  # второе искомое слово
  \W+ (?:\w+\W+){0,5}?   # до пяти слов
  word1                  # первое искомое слово
)\b
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Во втором регулярном выражении использован режим свободного форматирования и добавлены дополнительные символы пробела для повышения удобочитаемости. Во всем остальном эти два регулярных выражения совершенно идентичны. Диалект JavaScript не поддерживает режим свободного форматирования (если не используется библиотека XRegExp), но другие диалекты предоставляют такую возможность. В рецептах 3.5 и 3.7 показано, как можно добавить эти регулярные выражения в форму поиска или в другой программный код.

Обсуждение

Это регулярное выражение содержит две инвертированные копии одного и того же шаблона, следующие друг за другом, и окружает их границами слова. Первое подвыражение совпадает со словом `word1` и следующим за ним словом `word2`, между которыми могут находиться от нуля до пяти других слов. Второе подвыражение совпадает с теми же словами `word1` и `word2`, но следующими в обратном порядке.

В обоих подвыражениях присутствует минимальный квантификатор `‹{0,5}?›`. Он заставляет регулярное выражение соответствовать минимально возможному числу слов, расположенных между искомыми словами. Если попытаться сопоставить регулярное выражение с испытуемым текстом `word1 word2 word2`, оно совпадет с фрагментом `word1 word2`, потому что между первым и последним словами в совпадении присутствует минимальное (ноль) число слов. Если появится необходимость настроить расстояние между искомыми словами, можно заменить 0 и 5 внутри квантификаторов на нужные значения. Например, если заменить их значениями `‹{1,15}?›`, это позволит отыскивать требуемые слова, между которыми могут находиться до 15 слов и которые обязательно должны отделяться хотя бы одним словом.

Конструкции сокращенной формы записи символьных классов, которые используются для сопоставления с символами слова и символами, не являющимися символами слова (`‹\w›` и `‹\W›` соответственно), следуют причудливому определению регулярных выражений относительно того, какие символы считаются символами слова (буквы, цифры и символ подчеркивания).

Варианты

Использование условных конструкций

Нередко одно и то же регулярное выражение можно записать несколькими способами. В этой книге мы старались удерживать баланс между переносимостью, краткостью, эффективностью и некоторыми другими характеристиками. Однако иногда даже далеко не идеальные решения могут служить наглядными примерами. В следующих регулярных выражениях демонстрируются альтернативные способы поиска слов, расположенных недалеко друг от друга. Мы не рекомендуем использовать их на практике, так как, хотя они и будут совпадать с теми же самыми фрагментами текста, но эту работу будут выполнять немного дольше. Кроме того, они будут работать не во всех диалектах регулярных выражений.

В первом регулярном выражении вместо простого объединения инвертированных шаблонов используется условная конструкция (рецепт 2.17), выясняющая наличие совпадения со словом `word1` или `word2` в конце регулярного выражения. Условная конструкция проверяет, участвовала

ли первая сохраняющая группа в сопоставлении; это означает, что совпадение началось со слова word2:

```
\b(?:word1|(word2))\W+(?:\w+\W+){0,5}?(?(1)word1|word2)\b
```

Параметры: нет
Диалекты: .NET, PCRE, Perl, Python

Следующая версия также использует условную конструкцию, чтобы определить, совпадение с каким словом следует искать в конце, но добавляет еще две особенности регулярных выражений:

```
\b(?:(?<w1>word1)|(?<w2>word2))\W+(?:\w+\W+){0,5}?(?(w2)(?&w1)|(?&w2))\b
```

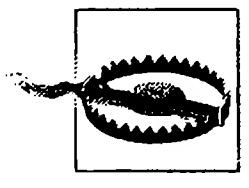
Параметры: нет
Диалекты: PCRE 7, Perl 5.10

Здесь первые вхождения ‹word1› и ‹word2› заключены в именованные сохраняющие группы, которые записываются как ‹(?<*name*>···)›. Это позволяет задействовать синтаксис *обращения к подпрограмме* ‹(?&*name*)› для повторного использования подвыражения обращением к нему по имени. Это далеко не то же самое, что обратная ссылка на именованную группу. Именованная обратная ссылка, такая как ‹\k<*name*>› (.NET, Java 7, XRegExp, PCRE 7, Perl 5.10) или ‹(?P=*name*)› (PCRE 4, Perl 5.10, Python), позволяет добавить повторное сопоставление с текстом, который уже совпал с именованной сохраняющей группой. Подпрограмма же, такая как ‹(?&*name*)›, повторно задействует фактический шаблон, заключенный в соответствующую сохраняющую группу. Здесь нельзя использовать обратные ссылки, потому что в данном случае они обеспечили бы сопоставление со словами, которые уже совпали. Подпрограммы внутри условной конструкции, находящейся в конце регулярного выражения, обеспечивают сопоставление со словом, совпадения с которым еще не было, избавляя от необходимости употреблять искомые слова еще раз. Это означает, что, если потребуется использовать это регулярное выражение для поиска других пар слов, достаточно будет изменить их лишь в одном месте.

В Ruby 1.9 именованные подпрограммы поддерживаются в виде конструкции ‹\g<*name*>›, но, так как в этом диалекте не поддерживаются условные конструкции, в нем нельзя использовать регулярные выражения, представленные выше. PCRE 4 стала первой библиотекой регулярных выражений, в которой появилась поддержка именованных подпрограмм, но поначалу в ней использовался синтаксис ‹(?P>*name*)›, который ныне вытеснен Perl-совместимым синтаксисом ‹(?&*name*)›, добавленным в Perl 5.10 и PCRE 7. В версию PCRE 7.7 в качестве дополнительной возможности была добавлена поддержка синтаксиса подпрограмм из Ruby 1.9.

Сопоставление с тремя или более словами, расположенными поблизости

Экспоненциальный рост числа перестановок. Сопоставление с двумя словами, расположенными поблизости, является достаточно простой задачей. В конце концов, существует всего два способа их взаимного расположения. Но что если потребуется отыскать три слова, которые могут следовать в произвольном порядке? В этом случае имеется шесть возможных вариантов их взаимного расположения (пример 5.1). Количество вариантов взаимного расположения слов равно $n!$, или произведению последовательности целых чисел от 1 до n (n факториал). Для четырех слов возможно 24 варианта их расположения. Когда вы доберетесь до 10 слов, число вариантов превысит несколько миллионов. Просто нереально реализовать сопоставление более чем с несколькими словами, расположенными поблизости друг от друга, используя приемы регулярных выражений, обсуждавшиеся до сих пор.



Понятия, которые представлены далее в этом разделе, относятся к одним из самых сложных в этой книге. Поэтому не надо плохо думать о своих умственных способностях, если вам не удастся разобраться во всем с первого прочтения.

Некрасивое решение. Один из способов решить эту проблему заключается в том, чтобы повторять группу, совпадающую с искомыми словами или любым другим словом (после того как будет обнаружено совпадение с искомым словом), и затем с помощью условных конструкций препятствовать успешному завершению попытки сопоставления, пока не будут найдены все искомые слова. Ниже приводится пример сопоставления с тремя словами, следующими в любом порядке, которые могут разделять до пяти слов:

```
\b(?:(?>(word1)|(word2)|(word3)|(?(1)|(?(2)|(?(3)|(?!))))\w+)\b\W*?){3,8}↵
(?(1)(?(2)(?(3)|(?!))|(?!))|(?!))
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, PCRE, Perl

Пример 5.1. Количество возможных перестановок

```
Два значения:
    [ 12, 21 ]
    = 2 возможных перестановки

Три значения:
     [ 123, 132,
       213, 231,
       312, 321 ]
    = 6 возможных перестановок
```

```
Четыре значения:
     [ 1234, 1243, 1324, 1342, 1423, 1432,
       2134, 2143, 2314, 2341, 2413, 2432,
       3124, 3142, 3214, 3241, 3412, 3421,
       4123, 4132, 4213, 4231, 4312, 4321 ]
    = 24 возможных перестановки

Факториалы:
    2! = 2 × 1                                          =       2
    3! = 3 × 2 × 1                                      =       6
    4! = 4 × 3 × 2 × 1                                  =      24
    5! = 5 × 4 × 3 × 2 × 1                              =     120
    . . .
    10! = 10 × 9 × 8 × 7 × 6 × 5 × 4 × 3 × 2 × 1 = 3628800
```

Ниже приводится еще одно регулярное выражение, в котором атомарная группировка (рецепт 2.14) была заменена стандартной несохраняющей группировкой с целью обеспечить поддержку диалекта Python:

```
\b(?:(?:(word1)|(word2)|(word3)|(?(1)|(?(2)|(?(3)|(?!))))\w+)\b\W*?){3,8}↵
(?(1)(?(2)(?(3)|(?!))|(?!))|(?!))
```

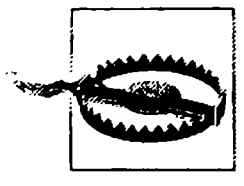
Параметры: нечувствительность к регистру символов
Диалекты: .NET, PCRE, Perl, Python

Квантификаторы ‹{3,8}›, используемые в приведенных выше регулярных выражениях, просто учитывают три обязательных слова и тем самым допускают наличие от нуля до пяти слов между ними. Пустые негативные опережающие проверки, такие как ‹(?!)›, никогда не будут обнаруживать совпадения и используются, чтобы блокировать дальнейшее сопоставление, пока не будет найдено совпадение хотя бы с одним обязательным словом. Логика управления сопоставлением реализована на базе двух наборов вложенных условных конструкций. Первый набор, используя конструкцию ‹\w+›, препятствует совпадению с любым из прежних слов, пока не будет найдено совпадение хотя бы с одним обязательным словом. Второй набор условных конструкций в конце выражения вынуждает механизм регулярных выражений производить возврат или терпеть неудачу, если не было найдено совпадение со всеми обязательными словами.

Мы привели краткий обзор того, как это все работает, но вместо того чтобы дальше погружаться в детали и выяснять, как можно добавить дополнительные обязательные слова, рассмотрим улучшенную реализацию, которая обеспечивает поддержку большего числа диалектов регулярных выражений и использует некоторые хитрости.

Использование пустых обратных ссылок. Некрасивое решение вполне работоспособно, но оно легко может победить в конкурсе самых малопонятных регулярных выражений на звание самого неудобочитаемого и сложного в сопровождении. Оно станет еще хуже, если попытаться добавить в него дополнительные обязательные слова.

К счастью, регулярные выражения предоставляют одну зацепку, которую можно использовать, чтобы упростить выражение; при этом она добавляет поддержку диалектов Java и Ruby (ни один из которых не поддерживает условные конструкции).



Прием, описываемый в этом разделе, должен с особой осторожностью использоваться в окончательных версиях приложений. Мы используем здесь особенности поведения регулярных выражений, недокументированные в большинстве библиотек.

```
\b(?:(?>word1()|word2()|word3()|(?>\1|\2|\3)\w+)\b\W*?){3,8}\1\2\3
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
\b(?:(?:word1()|word2()|word3()|(?:\1|\2|\3)\w+)\b\W*?){3,8}\1\2\3
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

С помощью этой конструкции легко можно добавлять обязательные слова. Ниже приводится пример, позволяющий находить четыре обязательных слова, следующие в произвольном порядке, которые могут разделять до пяти слов:

```
\b(?:(?>word1()|word2()|word3()|word4()|↵
(?>\1|\2|\3|\4)\w+)\b\W*?){4,9}\1\2\3\4
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Ruby

```
\b(?:(?:word1()|word2()|word3()|word4()|↵
(?:\1|\2|\3|\4)\w+)\b\W*?){4,9}\1\2\3\4
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Эти регулярные выражения специально используют пустые сохраняющие группы после каждого обязательного слова. Так как попытка сопоставления с обратной ссылкой, такой как ‹\1›, будет терпеть неудачу, если соответствующая ей сохраняющая группа еще не участвовала в сопоставлении, то обратные ссылки на пустые сохраняющие группы могут использоваться для управления процессом перемещения механизма регулярных выражений по шаблону, как это делают условные конструкции в регулярных выражениях, продемонстрированных выше. Если соответствующая группа уже участвовала в сопоставлении, то когда механизм регулярных выражений достигнет обратной ссылки на эту группу, он просто обнаружит совпадение с пустой строкой и двинется дальше.

Группировка ‹(?>\1|\2|\3)› в данном примере препятствует совпадению слова с ‹\w+›, пока не будет найдено хотя бы одно совпадение с обязательным словом. В конце регулярного выражения обратные ссылки повторяются еще раз, чтобы воспрепятствовать окончательному совпадению, пока не будут найдены все три обязательных слова.

Диалект Python не поддерживает атомарную группировку, поэтому в примерах, где Python перечислен в списке поддерживаемых диалектов, атомарные группировки заменены обычными несохраняющими группами. Это приводит к снижению эффективности регулярного выражения, но не влияет на результат сопоставления. Самая внешняя группировка не может быть атомарной ни в одном из диалектов, потому что для выполнения поставленной задачи механизму регулярных выражений требуется иметь возможность выполнять возвраты внутри внешней группы, если обратные ссылки в конце регулярного выражения терпят неудачу при сопоставлении.

В JavaScript применяются свои правила для работы с обратными ссылками. Хотя диалект JavaScript поддерживает все синтаксические конструкции, присутствующие в версии этого шаблона для Python, он имеет два ограничения, препятствующие возможности использовать эту хитрость, как в других диалектах. Первая проблема состоит в том, что обратная ссылка на сохраняющую группу будет обнаруживать совпадение, даже если она еще не участвовала в сопоставлении. Спецификация на JavaScript требует, чтобы такие обратные ссылки соответствовали пустой строке или, говоря другими словами, они всегда обнаруживают совпадение. Практически во всех других диалектах все происходит с точностью до наоборот: они никогда не обнаруживают совпадение, в результате чего механизм регулярных выражений вынужден производить возвраты, либо пока не потерпит неудачу совпадение со всем регулярным выражением, либо пока группы, на которые они ссылаются, не примут участие в сопоставлении и не обеспечат возможность совпадения для обратных ссылок.

Вторая проблема в диалекте JavaScript связана со значениями, которые запоминаются сохраняющими группами, вложенными в повторяющуюся внешнюю группу, например `‹((a)|(b))+›`. В большинстве диалектов регулярных выражений значениями, запоминаемыми сохраняющими группами, находящимися внутри повторяющейся внешней группы, являются совпадения, обнаруженные в последней попытке сопоставления. Так, если при попытке сопоставления выражение `‹(?:(a)|(b))+›` обнаружило совпадение с фрагментом ab, значением первой обратной ссылки будет символ a. Однако согласно спецификации на JavaScript, значения обратных ссылок на вложенные группы должны сбрасываться всякий раз, когда выполняется очередное повторение внешней группы. По этой причине выражение `‹(?:(a)|(b))+›` также будет совпадать с фрагментом ab, но по окончании сопоставления обратная ссылка 1 будет ссылаться на сохраняющую группу, не участвовавшую в сопоставлении, и в JavaScript будет совпадать с пустой строкой в пределах самого регулярного выражения и возвращать значение `undefined`, например в массиве, возвращаемом методом *regexp*`.exec()`.

Любого из этих отличий в поведении диалекта JavaScript достаточно, чтобы избегать возможности имитировать условные конструкции с помощью пустых сохраняющих групп, как было описано выше.

Множество слов на любом расстоянии друг от друга

Если необходимо просто проверить наличие некоторых слов в испытуемом тексте, не принимая во внимание их близость, можно воспользоваться опережающей проверкой, чтобы реализовать поиск в одной операции.

Во многих случаях гораздо проще и эффективнее выполнять несколько операций поиска, по одной для каждого искомого слова, просто следя за тем, чтобы все проверки дали положительный результат.

```
^(?=.*?\bword1\b)(?=.*?\bword2\b).*
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
^(?=[\s\S]*?\bword1\b)(?=[\s\S]*?\bword2\b)[\s\S]*
```

Параметры: нечувствительность к регистру символов (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Эти регулярные выражения совпадают со всем испытуемым текстом, если внутри него присутствуют все искомые слова. В противном случае никаких совпадений обнаружено не будет. Программисты на JavaScript не смогут использовать первую версию без применения библиотеки XRegExp, потому что стандартный JavaScript не поддерживает режим «точке соответствуют границы строк».

Эти регулярные выражения можно реализовать, следуя примеру в рецепте 3.6. Перед его использованием необходимо заменить ‹word1› и ‹word2› словами, которые требуется отыскать. Если необходимо проверить наличие большего числа слов, можно добавить в начало регулярного выражения столько опережающих проверок, сколько потребуется. Например, выражение ‹^(?=.*?\bword1\b)(?=.*?\bword2\b)(?=.*?\bword3\b).*› отыскивает три слова.

См. также

В рецепте 5.5 объясняется, как отыскать любое слово, за которым не следует указанное слово. В рецепте 5.6 описывается, как отыскать любое слово, которому не предшествует указанное слово.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется,

как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.14 описывается атомарная группировка. В рецепте 2.17 демонстрируется применение условных конструкций.

5.8. Поиск повторяющихся слов

Задача

Вы редактируете некоторый документ, и вам потребовалось проверить наличие ошибочно повторяющихся слов. При поиске не должны учитываться различия в регистре символов, такие как «The the». Кроме того, требуется обеспечить возможность разделения слов разным числом пробелов, даже если при этом слова будут переноситься на другую строку. Однако они не должны расцениваться как повторяющиеся при наличии знаков пунктуации между одинаковыми словами.

Решение

Обратная ссылка соответствует тому, что уже совпало раньше, благодаря чему она является ключевым ингредиентом данного рецепта:

```
\b([A-Z]+)\s+\1\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если предполагается использовать это выражение, чтобы сохранить первое слово и удалить все последующие дубликаты, следует заменить все совпадения с обратной ссылкой 1. Другой подход заключается в том, чтобы выделить совпадения, окружив их другими символами (например, тегами HTML), чтобы при последующем просмотре их легко можно было идентифицировать. В рецепте 3.15 показано, как использовать обратные ссылки в замещающем тексте, который потребуется для реализации любого из этих подходов.

Если требуется просто отыскать повторяющиеся слова, чтобы самому проверить, действительно ли это ошибка, можно воспользоваться рецептом 3.7, где приводится программный код, необходимый для этого. Текстовый редактор или инструмент, подобный grep, аналогичный тем, что перечислены в разделе «Инструменты для работы с регулярными выражениями» главы 1, помогут вам отыскать повторяющиеся слова, а доступность окружающего текста позволит определить, действительно ли повторение слов является ошибкой, которую следует исправить.

Обсуждение

Чтобы обнаружить повторное совпадение с тем, что уже совпало ранее, необходимы сохраняющая группа и обратная ссылка. Нужно просто поместить в сохраняющую группу то, что может совпадать более одного раза, и затем добавить сопоставление с обратной ссылкой. Этот прием

работает иначе, чем простое повторение элемента или группы с помощью квантификатора. Рассмотрим различия между простыми регулярными выражениями ‹(\w)\1› и ‹\w{2}›. В первом выражении для обнаружения повторного совпадения с тем же самым символом слова используются сохраняющая группа и обратная ссылка, тогда как во втором выражении применение квантификатора обеспечивает совпадение с любыми двумя символами слова. Подробнее об обратных ссылках рассказывается в рецепте 2.10.

Вернемся к нашей задаче. Решение, представленное в этом рецепте, отыскивает повторяющиеся слова, состоящие только из символов от A до Z и от a до z (поскольку включен режим нечувствительности к регистру символов). Чтобы обеспечить возможность поиска повторяющихся слов, содержащих символы из других алфавитов, можно использовать категорию Юникода Letter ‹\p{L}›, если диалект регулярных выражений поддерживает такую возможность (раздел «Категории Юникода» в рецепте 2.7).

Элементу ‹\s+›, расположенному между сохраняющей группой и обратной ссылкой, соответствует любое число пробельных символов, таких как пробел, табуляция или разрыв строки. Если требуется ограничить набор символов, которые могут разделять повторяющиеся слова, горизонтальными пробельными символами (то есть исключить из совпадения разрывы строк), нужно заменить ‹\s+› на ‹[•\t\xA0]›. Это позволит избежать совпадения с повторяющимися словами, расположенными в разных строках текста. Комбинация ‹\xA0› в символьном классе соответствует неразрывному пробелу, который встречается в текстах, скопированных из веб-страниц (многие веб-разработчики используют мнемонику   для вставки неразрывных пробелов в содержимое веб-страниц). Диалекты PCRE 7.2 и Perl 5.10 включают метасимвол ‹\h›, представляющий сокращенную форму записи символьного класса, который предпочтительнее в данной ситуации, так как он специально создан, чтобы соответствовать горизонтальному пробельному символу, а также некоторым другим необычным пробельным символам.

Наконец, границы слова в начале и в конце регулярного выражения гарантируют, что совпадения не будут обнаруживаться в середине других слов (например, «this thistle»).

Следует заметить, что повторяющиеся слова в тексте не всегда являются ошибкой, поэтому простое удаление их без визуального просмотра может оказаться неправильным. Например, в разговорном английском языке допустимыми считаются такие конструкции, как «that that» или «had had». Омонимы, названия, звукоподражательные слова (такие как «oink oink» или «ha ha») и некоторые другие конструкции иногда образуются повторяющимися словами. Поэтому в большинстве случаев необходимо визуально исследовать каждое совпадение.

Варианты

Решение, представленное выше, специально было максимально упрощено. Но из-за этой простоты не учитываются некоторые частные случаи:

- Повторяющиеся слова, включающие буквы с акцентами и другими диакритическими знаками, такие как «café café» или «naïve naïve».
- Повторяющиеся слова, включающие дефисы, апострофы или закрывающие одиночные кавычки, такие как «co-chair co-chair», «don't don't» или «rollin' rollin'».
- Повторяющиеся слова, состоящие из букв нелатинского алфавита, такие как русские слова «друзья друзья».

Необходимость решения этих проблем вынуждает нас отказаться от метасимвола ‹\b› границы слова, который использовался выше, чтобы гарантировать совпадение с целым словом. Неработоспособность ‹\b› в перечисленных только что ситуациях объясняется двумя причинами. Во-первых, дефисы и апострофы не являются символами слова, поэтому граница слова не будет обнаружена между пробельным символом или знаком пунктуации, отделяющим слова, и дефисом или апострофом, находящимся в начале или в конце слова. Во-вторых, в некоторых диалектах метасимвол ‹\b› не поддерживает Юникод (раздел «Символы слов» в рецепте 2.6), поэтому он не всегда будет работать правильно, если в испытуемом тексте имеются буквы, отличные от букв A–Z латинского алфавита без диакритических знаков.

По этим причинам, чтобы гарантировать совпадение только с целым словом, вместо ‹\b› следует использовать опережающие и ретроспективные проверки (рецепт 2.16). Для поддержки букв и диакритических знаков из любых алфавитов необходимо также использовать категории Юникода ‹\p{L}› и ‹\p{M}› (рецепт 2.7):

```
(?<![\p{L}\p{M}\-'\u2019])([\-'\u2019]?(?:[\p{L}\p{M}][\-'\u2019]?)+)↵
\s+\1(?![\p{L}\p{M}\-'\u2019])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, Ruby 1.9

Несмотря на то что категория ‹\p{L}› соответствует любым буквам в любом регистре, необходимо использовать параметр «нечувствительность к регистру символов», потому что при сохранении совпадения в обратной ссылке ‹\1› может использоваться регистр символов, отличный от регистра символов в исходном совпавшем слове.

Последовательности ‹\u2019› в регулярном выражении совпадают с закрывающей одиночной кавычкой ('). В диалектах Perl и PCRE используется другой синтаксис сопоставления с отдельными кодовыми пунктами Юникода, поэтому для поддержки этих диалектов регулярное выражение необходимо немного изменить:

```
(?<![\p{L}\p{M}\-'\x{2019}])([\-'\x{2019}]?(?:[\p{L}\p{M}]↵
[\-'\x{2019}]?)+)\s+\1(?![\p{L}\p{M}\-'\x{2019}])
```

Параметры: нечувствительность к регистру символов
Диалекты: Java 7, PCRE, Perl

Ни одно из этих регулярных выражений не может использоваться в JavaScript, Python или Ruby 1.8, так как в этих диалектах отсутствует поддержка категорий Юникода, таких как ‹\p{L}›. Кроме того, в JavaScript и Ruby 1.8 отсутствует поддержка ретроспективных проверок.

Ниже приводится несколько примеров повторяющихся слов, совпадающих с представленными выше регулярными выражениями:

- The the
- café café
- друзья друзья
- don't don't
- rollin' rollin'
- O'Keeffe's O'Keeffe's
- co-chair co-chair
- devil-may-care devil-may-care

Ниже приводится несколько примеров строк, не совпадающих с регулярными выражениям:

- hello, hello
- 1000 1000
- - -
- test''ing test''ing
- one--two one--two

См. также

В рецепте 5.9 демонстрируется, как отыскивать повторяющиеся строки.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.7 рассказывается о сопоставлении с символами Юникода. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

5.9. Удаление повторяющихся строк

Задача

Имеется некоторый файл журнала, результат выполнения запроса к базе данных или какой-нибудь другой файл или текст с повторяющимися строками. Необходимо с помощью текстового редактора или похо-

жего инструмента удалить все дубликаты, оставив единственную строку из каждой группы повторений.

Решение

Существуют различные программы (включая утилиту `uniq` командной строки в системе UNIX и команду `Get-Unique` оболочки PowerShell в системе Windows), способные помочь удалить дубликаты строк, встречающиеся в файле или в тексте. В следующем разделе будут представлены три варианта решения задачи, основанные на использовании регулярных выражений, которые могут быть особенно полезны при попытке решить эту задачу в текстовом редакторе, обеспечивающем поддержку регулярных выражений и операции поиска с заменой.

При программировании второе и третье решения использовать не следует, поскольку они неэффективны в сравнении с другими доступными средствами, такими как использование хешей[1] для хранения уникальных строк. Однако первое решение (которое требует, чтобы строки были отсортированы заранее, что позволит удалить не только рядом стоящие дубликаты) может быть вполне применимо, так как отличается простотой и высокой скоростью работы.

Решение 1: сортировка строк и удаление смежных дубликатов

Если есть возможность отсортировать строки в файле или в тексте, чтобы все повторяющиеся строки оказались следующими друг за другом, ее необходимо использовать при условии, что не требуется сохранить первоначальный порядок следования строк. Это позволит применить для удаления дубликатов более простую и эффективную по сравнению с другими операцию поиска с заменой.

Чтобы удалить дубликаты строк после сортировки, можно использовать следующее регулярное выражение и замещающий текст:

```
^(.*)(?:(?:\r?\n|\r)\1)+$
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст:

```
$1
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\1
```

Диалекты замещающего текста: Python, Ruby

---

[1] Ассоциативные массивы. – *Прим. перев.*

Для совпадения с двумя или более дубликатами строк, следующими друг за другом, в этом регулярном выражении используются сохраняющая группа и обратная ссылка (помимо других ингредиентов). В замещающем тексте используется обратная ссылка, которая вставляет первую строку обратно в текст. В рецепте 3.15 приводится пример программного кода, который можно использовать для реализации этого решения.

Решение 2: оставить в неотсортированном файле последнее вхождение каждой повторяющейся строки

В случае использования текстового редактора, в котором отсутствует возможность сортировать строки, или если важно сохранить оригинальный порядок следования строк, удалить дубликаты поможет следующее решение, даже если повторяющиеся строки разделены другими строками:

```
^([^\r\n]*)(?:\r?\n|\r)(?=.*^\1$)
```

Параметры: точке соответствуют границы строк, символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Ниже приводится то же самое решение для стандартного диалекта JavaScript, которое не требует включения режима «точке соответствуют границы строк»:

```
^(.*)(?:\r?\n|\r)(?=[\s\S]*^\1$)
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст:

(Пустая строка, то есть ничего.)

Диалекты замещающего текста: не применимо в данном случае

Решение 3: оставить в неотсортированном файле первое вхождение каждой повторяющейся строки

Если требуется сохранить первое вхождение из каждой группы повторяющихся строк, необходимо использовать иной подход. Сначала приведем регулярное выражение и замещающий текст, которые мы будем использовать:

```
^([^\r\n]*)$(.*?)(?:(?:\r?\n|\r)\1$)+
```

Параметры: точке соответствуют границы строк, символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Чтобы преодолеть несовместимость этого регулярного выражения со стандартным диалектом JavaScript из-за отсутствия в нем поддержки

режима «точке соответствуют границы строк», необходимо внести пару изменений:

```
^(.*)$([\s\S]*?)(?:(?:\r?\n|\r)\1$)+
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст:

```
$1$2
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\1\2
```

Диалекты замещающего текста: Python, Ruby

В отличие от регулярных выражений в решениях 1 и 2, эта версия не может удалить все дубликаты строк в ходе одной операции поиска с заменой. Вам потребуется постоянно повторять операцию «заменить все», пока регулярное выражение не перестанет обнаруживать совпадения, то есть пока в тексте не останется повторяющихся строк. Дополнительные подробности приводятся в разделе «Обсуждение» в этом рецепте.

Обсуждение

Решение 1: сортировка строк и удаление смежных дубликатов

Это регулярное выражение удаляет все повторяющиеся строки, следующие друг за другом, за исключением первой. Оно не удаляет дубликаты, если между ними присутствуют другие строки. Рассмотрим порядок работы выражения поближе.

Во-первых, символ крышки ‹^› в начале регулярного выражения совпадает с началом строки. Обычно он совпадает только с началом всего испытуемого текста, поэтому необходимо включить режим «символам ^ и $ соответствуют границы строк» (в рецепте 3.4 показано, как устанавливать различные режимы в программном коде). Затем конструкция ‹.*› внутри сохраняющих круглых скобок совпадает со всем содержимым одной строки (даже если строка пустая), а значение сохраняется как обратная ссылка 1. Чтобы не возникало ошибок, необходимо выключить режим «точке соответствуют границы строки», в противном случае точка со звездочкой совпадут со всем текстом.

Внутри внешней несохраняющей группы мы использовали конструкцию ‹(?:\r?\n|\r)›, которой соответствуют разделители строк в текстовых файлах, используемые в Windows/MS-DOS (‹\r\n›), в UNIX/Linux/BSD/OS X (‹\n›) или устаревшей версии Mac OS (‹\r›). Затем обратная ссылка ‹\1› пытается найти совпадение со строкой, сопоставление с которой только что было завершено. Если в этой позиции не будет найдена та же самая строка, попытка сопоставления с регулярным выражением

терпит неудачу и механизм регулярных выражений перемещается в следующую позицию. Если совпадение обнаруживается, квантификатор ‹+› выполняет повтор группы (составленной из последовательности разрыва строки и обратной ссылки 1), чтобы попытаться найти совпадение с дополнительными дубликатами строк.

Наконец, мы использовали знак доллара в конце регулярного выражения, чтобы проверить совпадение с концом строки в тексте. Это гарантирует совпадение только с идентичными строками, но не со строками, начинающимися с тех же символов, которые составляют содержимое предыдущей строки.

Поскольку выполняется операция поиска с заменой, из текста удаляется каждое совпавшее вхождение (включая оригинальную строку и разделители строк). Замещающий текст содержит обратную ссылку 1, которая вставляет оригинальную строку обратно в текст.

Решение 2: оставить в неотсортированном файле последнее вхождение каждой повторяющейся строки

Это решение несколько отличается от решения 1, которое отыскивает дубликаты строк, только если они следуют непосредственно друг за другом. Во-первых, в версии для диалектов, отличных от JavaScript, в данном решении точка внутри сохраняющей группы заменена конструкцией ‹[^\r\n]› (совпадает с любым символом, кроме разрыва строки) и включен режим «точке соответствуют границы строк». Сделано это потому, что далее в регулярном выражении точка используется с целью обеспечить совпадение с любым символом, включая разрывы строк. Во-вторых, для поиска дубликатов строки, которые могут встретиться далее в испытуемом тексте, была добавлена опережающая проверка. Поскольку опережающая проверка не поглощает символы, регулярное выражение всегда будет совпадать с единственной строкой (вместе с завершающим ее разрывом строки), которая повторно встречается ниже в испытуемом тексте. Замена всех совпадений пустыми строками удалит все дубликаты строк, при этом нетронутыми останутся только последние вхождения.

Решение 3: оставить в неотсортированном файле первое вхождение каждой повторяющейся строки

Ретроспективная проверка поддерживается не так широко, как опережающая, а там, где поддерживается, она может оказаться не в состоянии заглядывать назад настолько далеко, как это необходимо. Поэтому регулярное выражение в решении 3 концептуально отличается от регулярного выражения в решении 2. Вместо поиска строк, которые уже встречались выше (что можно было бы сравнить с тактикой решения 2), это регулярное выражение совпадает со строкой, с первым дубликатом этой строки, который встречается ниже в тексте, и всеми строками между ними. Оригинальная строка сохраняется как обратная ссылка 1,

а строки между дубликатами (если таковые вообще имеются) – как обратная ссылка 2. Операция замещения совпадения обеими обратными ссылками 1 и 2 возвращает в текст те части совпадения, которые требуется сохранить, и отбрасывает строку-дубликат и предшествующий ей разрыв строки.

Этот альтернативный подход таит в себе две проблемы. Во-первых, так как каждое совпадение с дубликатами строк может включать строки, находящиеся между ними, вполне вероятно, что среди них окажутся другие строки, которые также могут повторяться, но будут пропущены в процессе выполнения операции замены. Во-вторых, если одна и та же строка встречается в тексте более двух раз, то регулярное выражение сначала совпадет с первым и со вторым дубликатами, но все последующие дубликаты будут восприниматься им как другие наборы дубликатов, которые соответствуют регулярному выражению по мере продвижения по тексту. Вследствие этого операция замены в лучшем случае удалит только каждое второе вхождение той или иной строки. Чтобы ликвидировать эти проблемы и гарантировать удаление всех дубликатов, необходимо выполнять операцию поиска с заменой для всего испытуемого текста, пока регулярное выражение не перестанет обнаруживать совпадения. Рассмотрим, как будет работать это регулярное выражение применительно к следующему тексту:

```
value1
value2
value2
value3
value3
value1
value2
```

Для удаления всех дубликатов строк из этого текста потребуется выполнить три прохода. Результаты каждого из проходов приводятся в табл. 5.1.

Таблица 5.1. Результаты операции поиска с заменой после каждого прохода

Проход 1	Проход 2	Проход 3	Окончательный текст
value1	value1	value1	value1
value2	value2	value2	value2
value2	~~value2~~	value3	value3
value3	value3	~~value2~~	
value3	~~value3~~		
~~value1~~	value2		
value2			
Одно совпадение/замена	Два совпадения/замены	Одно совпадение/замена	Дубликатов не осталось

См. также

В рецепте 5.8 демонстрируется, как отыскивать повторяющиеся слова.

В рецепте 3.19 приводятся примеры программного кода, разбивающего строку с использованием регулярного выражения, демонстрирующие альтернативный подход к удалению повторяющихся строк, где регулярное выражение играет лишь вспомогательную роль. Воспользовавшись регулярным выражением, совпадающим с разрывами строк (таким как `‹\r?\n|\r›`) в качестве разделителя для операции разбиения, легко можно получить список всех строк, имеющихся в испытуемом тексте. Затем можно выполнить цикл по этому списку и выбрать из него уникальные строки с помощью объекта-хеша, отвергающего дубликаты прежде встретившихся строк.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.21 демонстрируется, как вставлять текст, совпавший с сохраняющей группой, в текст замены.

5.10. Совпадение с полными строками, содержащими определенное слово

Задача

Необходимо обеспечить совпадение с любыми строками, содержащими слово `error` в любом месте внутри этих строк.

Решение

```
^.*\berror\b.*$
```

Параметры: нечувствительность к регистру символов, символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Часто бывает удобно находить совпадения с целыми строками для их отбора или удаления. Мы начнем наши попытки отыскать совпадение с любой строкой, содержащей слово `error`, с выражения `‹\berror\b›`. Как описывалось в рецепте 2.6, метасимволы границы слова, находящиеся

с обоих концов регулярного выражения, гарантируют, что совпадение будет обнаруживаться с последовательностью символов «error», только когда она представляет собой отдельное слово.

Чтобы расширить область совпадения с регулярным выражением до полной строки, добавим конструкцию ‹.*› с обоих концов. Последовательности точки со звездочкой соответствуют ноль или более символов внутри текущей строки. Звездочка является максимальным квантификатором, поэтому она обеспечит совпадение максимально возможной длины. Первой точке со звездочкой будет соответствовать часть строки до последнего вхождения слова «error», а второй точке со звездочкой – все остальные символы, не являющиеся разрывами строк, следующие за искомым словом.

Наконец, поместим символ крышки и знак доллара в начало и в конец регулярного выражения соответственно, чтобы обеспечить совпадение с полной строкой. Строго говоря, якорный метасимвол ‹$› в конце выражения не нужен, потому что точка и максимальная звездочка всегда обнаруживают совпадение только до конца строки. Однако знак доллара не повредит и сделает регулярное выражение немного более очевидным. Добавление якорных метасимволов в регулярные выражения там, где это уместно, иногда может помочь избежать неожиданностей, поэтому этот прием стоит взять на вооружение. Следует заметить, что, в отличие от знака доллара, символ крышки в начале регулярного выражения не всегда бывает избыточен, так как он гарантирует, что регулярное выражение будет совпадать только с полными строками, даже когда по каким-то причинам поиск начинается с середины строки.

Не следует забывать, что три ключевых метасимвола, используемые для ограничения совпадения с единственной строкой (якорные метасимволы ‹^› и ‹$› и точка), не имеют фиксированного значения. Чтобы переориентировать их на работу со строками, следует включить режим, позволяющий метасимволам ^ и $ совпадать с границами строк, а режим, когда точке соответствуют границы строк, следует отключить. В рецепте 3.4 показано, как переключать эти режимы в программном коде. В случае использования диалектов JavaScript или Ruby одним режимом, который надо контролировать, становится меньше, потому что в JavaScript отсутствует режим, позволяющий точке совпадать с границами строк, а в Ruby символ крышки и знак доллара всегда совпадают с границами строк.

Варианты

Чтобы отыскать строки, содержащие любое из нескольких слов, можно использовать конструкцию выбора:

```
^.*\b(one|two|three)\b.*$
```

Параметры: нечувствительность к регистру символов, символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Это регулярное выражение совпадает с любой строкой, содержащей хотя бы одно из трех слов: «one», «two» или «three». Круглые скобки, окружающие слова, служат двум целям. Во-первых, они ограничивают область действия конструкции выбора и, во-вторых, сохраняют в обратной ссылке 1 слово, которое фактически было встречено в строке. Если в строке присутствует более чем одно из этих слов, в обратной ссылке будет сохранено слово, стоящее ближе к правому концу строки. Это обусловлено тем, что квантификатор звездочка, расположенный перед круглыми скобками, является максимальным и будет расширять совпадение с точкой до тех пор, пока это возможно. Если сделать звездочку минимальной, как в выражении `‹^.*?\b(one|two|three)\b.*$›`, обратная ссылка 1 будет содержать слово из списка, стоящее ближе к левому концу строки.

Чтобы отыскать строки, которые должны содержать сразу несколько искомых слов, можно использовать опережающую проверку:

```
^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).+$
```

Параметры: нечувствительность к регистру символов, символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы обеспечить совпадение со строками, содержащими в себе три обязательных слова, в этом регулярном выражении используется позитивная опережающая проверка. Конструкция `‹.+›` в конце обеспечивает фактическое совпадение со строкой после того, как опережающие проверки определят, что строка соответствует требованиям.

См. также

Рецепт 5.11, где показано, как обеспечить совпадение с полными строками, не содержащими определенное слово.

Если не требуется совпадение с целыми строками, загляните в рецепт 5.1, где описывается, как отыскать определенное слово, и в рецепт 5.2, где объясняется, как отыскать любое из множества слов.

Рецепт 3.21, где приводятся примеры программного кода, выполняющего построчный поиск и упрощающего процесс поиска и идентификации строк в тексте.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

5.11. Совпадение с полными строками, не содержащими определенное слово

Задача

Необходимо обеспечить совпадение с полными строками, не содержащими слово `error`.

Решение

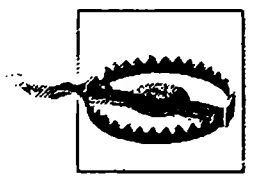
```
^(?:(?!\berror\b).)*$
```

Параметры: нечувствительность к регистру символов, символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Чтобы обеспечить совпадение со строкой, *не содержащей* некоторую последовательность символов, можно использовать негативную опережающую проверку (описывается в рецепте 2.16). Обратите внимание, что в этом регулярном выражении негативная опережающая проверка и точка объединены в повторяющуюся несохраняющую группу. Это обеспечивает проверку отсутствия совпадения с подвыражением ‹\berror\b› в каждой позиции строки. Якорные метасимволы ‹^› и ‹$›, находящиеся по обоим концам регулярного выражения, гарантируют совпадение со всей строкой целиком и дополнительно препятствуют группе, включающей негативную опережающую проверку, ограничиться проверкой только части строки.

Режимы, применяемые к этому регулярному выражению, определяют, будет ли совпадение включать в себя весь испытуемый текст или только одну строку. В случае когда включен режим «символам ^ и $ соответствуют границы строк» и выключен режим «точке соответствуют границы строк», это выражение действует, как только что было описано, и совпадает с единственной строкой. Если изменить эти два режима на обратные, регулярное выражение будет совпадать с любым текстом целиком, если в нем отсутствует слово «error».



Выполнение негативной опережающей проверки в каждой позиции строки или текста слишком неэффективно. Это решение может использоваться только в случаях, когда регулярное выражение является единственным доступным инструментом, например в приложении, которое нельзя изменить. При создании собственного приложения обратите внимание на рецепт 3.21, где представлено намного более эффективное решение, основанное на применении программного кода для построчного поиска.

См. также

Рецепт 5.10, где показано, как обеспечить совпадение с полными строками, *содержащими* определенное слово.

Рецепт 3.21, где приводятся примеры программного кода, выполняющего построчный поиск и упрощающего процесс поиска и идентификации строк в тексте.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

5.12. Удаление ведущих и завершающих пробельных символов

Задача

Необходимо удалить из строки ведущие и завершающие пробелы. Это может понадобиться, например, для подготовки данных, отправленных пользователем в составе веб-формы, к проверке с помощью одного из регулярных выражений, представленных в главе 4.

Решение

Чтобы сделать решение максимально простым и быстрым, лучше всего использовать два регулярных выражения, одно из которых будет удалять ведущие пробельные символы, а другое – завершающие.

Ведущие пробельные символы:

```
\A\s+
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^\s+
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Завершающие пробельные символы:

```
\s+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\s+$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Достаточно просто заменить пустыми строками совпадения, найденные одним из выражений, совпадающих с «ведущими пробельными символами», и одним из выражений, совпадающих с «завершающими пробельными символами». Как это сделать, демонстрируется в рецепте 3.14. В обоих случаях необходимо заменить только первое найденное совпадение, поскольку эти регулярные выражения совпадают со всеми ведущими или завершающими пробельными символами сразу.

Обсуждение

Удаление ведущих и завершающих пробельных символов – это достаточно простая, но часто встречающаяся задача. Каждое из представленных регулярных выражений делится на три части: символьный класс в сокращенной форме записи, совпадающий с любым пробельным символом (‹\s›), квантификатор, повторяющий символьный класс один или более раз (‹+›), и якорный метасимвол, проверяющий совпадение с началом или концом строки. Метасимволы ‹\A› и ‹^› совпадают с началом строки, а ‹\Z› и ‹$› – с концом.

Здесь представлено по два варианта регулярных выражений для поиска ведущих и завершающих пробельных символов из-за несовместимости диалектов Ruby и JavaScript. При использовании других диалектов можно выбирать любой из вариантов. Версии с метасимволами ‹^› и ‹$› будут неправильно работать в Ruby, потому что в этом диалекте данные метасимволы всегда совпадают с началом и концом любой строки. JavaScript не поддерживает якоря ‹\A› и ‹\Z›.

Многие языки программирования предоставляют специальную функцию, обычно называемую `trim` или `strip`, которая может удалять ведущие и завершающие пробельные символы. В табл. 5.2 показано, как использовать эту встроенную функцию или метод в различных языках программирования.

Таблица 5.2. Стандартные функции для удаления ведущих и завершающих пробельных символов

Язык(и) программирования	Пример использования
C#, VB.NET	*String*.Trim([*chars*])
Java, JavaScript	*string*.trim()
PHP	trim(*$string*)
Python, Ruby	*string*.strip()

В Perl отсутствует эквивалентная функция, но ее легко можно создать самому, используя регулярные выражения, представленные выше в этом рецепте:

```
sub trim {
    my $string = shift;
    $string =~ s/^\s+//;
    $string =~ s/\s+$//;
    return $string;
}
```

В JavaScript метод *string*.trim() появился совсем недавно. В старых браузерах (до версий Internet Explorer 9 и Firefox 3.5) его можно реализовать, как показано ниже:

```
// Добавить метод trim для браузеров, не имеющих его
if (!String.prototype.trim) {
    String.prototype.trim = function() {
        return this.replace(/^\s+/, "").replace(/\s+$/, "");
    };
}
```

В обоих языках программирования, Perl и JavaScript, метасимволу ‹\s› соответствует любой символ, определяемый стандартом Юникод как пробельный, в дополнение к символам пробела, табуляции, перевода строки и возврата каретки, которые наиболее часто рассматриваются как пробельные.

Варианты

В действительности существует масса различных способов записать регулярное выражение, которое поможет усекать строку. Однако при работе с длинными строками (когда производительность имеет самое большое значение) они неизменно оказываются медленнее, чем решение, опирающееся на использование двух простых регулярных выражений. Ниже приводятся некоторые из наиболее типичных альтернативных решений, с которыми можно столкнуться на практике. Все они написаны на JavaScript, а поскольку в стандартном JavaScript отсутствует режим «точке соответствуют границы строк», совпадение с любым единичным символом, включая разрывы строк, обеспечивается конструкцией ‹[\s\S]›. В других диалектах вместо этой конструкции можно использовать точку и включить режим «точке соответствуют границы строк».

```
string.replace(/^\s+|\s+$/g, "");
```

Это, пожалуй, наиболее часто используемое решение. Оно объединяет два простых регулярных выражения в одну конструкцию выбора (рецепт 2.8) и использует флаг /g («global» – глобальный), чтобы получить возможность заменить все, а не только первые совпадения (совпадений будет два – с ведущими и завершающими пробельными символами в строке). Это неплохое решение, но при работе с длинны-

ми строками оно медленнее решения, опирающегося на применение двух простых регулярных выражений, так как для каждой позиции приходится проверять две альтернативы.

```
string.replace(/^\s*([\s\S]*?)\s*$/, "$1")
```

Это регулярное выражение сопоставляется со всей строкой и сохраняет последовательность от первого до последнего непробельного символа (если таковые присутствуют) в обратной ссылке 1. После замены всей строки содержимым обратной ссылки 1 останется усеченная версия строки.

Этот подход концептуально проще, но минимальный квантификатор внутри сохраняющей группы заставляет механизм регулярных выражений выполнить массу дополнительной работы (то есть возвратов), что обусловливает более низкую скорость работы этого решения на длинных строках.

Давайте отступим на шаг назад и посмотрим, как это выражение работает. После того как в процессе сопоставления механизм регулярных выражений войдет в сохраняющую группу, минимальный квантификатор ‹*?› требует, чтобы символьный класс ‹[\s\S]› повторялся минимально возможное число раз. Поэтому при каждом проходе механизм регулярных выражений добавляет в совпадение один символ и останавливается, пытаясь сопоставить оставшуюся часть шаблона (‹\s*$›). Если эта попытка терпит неудачу, потому что где-то за текущей позицией в строке присутствует непробельный символ, механизм добавляет к совпадению с классом еще один символ, обновляет обратную ссылку и опять повторяет попытку сопоставления оставшейся части шаблона.

```
string.replace(/^\s*([\s\S]*\S)?\s*$/,  "$1")
```

Это решение напоминает предыдущее, но в нем из соображений производительности минимальный квантификатор заменен максимальным. Чтобы гарантировать совпадение сохраняющей группы вплоть до последнего непробельного символа, в конец ее потребовалось добавить метасимвол ‹\S›. Однако из-за того что регулярное выражение должно совпадать со строками, состоящими только из пробельных символов, вся сохраняющая группа сделана необязательной добавлением квантификатора ‹?› после нее.

Здесь максимальная звездочка в конструкции ‹[\s\S]*› повторяет шаблон «любой символ», пока не будет достигнут конец строки. Затем механизм регулярных выражений выполняет возвраты по одному символу, начиная с конца строки, пока не будет обнаружено совпадение со следующим метасимволом ‹\S› или пока в процессе возвратов не будет достигнут первый символ, совпавший с сохраняющей группой (после чего группа пропускается).

Если число завершающих пробельных символов не превышает длину текста, совпавшую к данному моменту, то в общем случае это решение оказывается быстрее предыдущего, где используется мини-

мальный квантификатор. Однако и это решение нельзя сравнить по производительности с решением, опирающимся на использование двух простых регулярных выражений.

```
string.replace(/^\s*(\S*(?:\s+\S+)*)\s*$/, "$1")
```

Это довольно часто встречающееся решение, но у него нет никаких преимуществ, потому что оно медленнее всех остальных решений, показанных выше. Это регулярное выражение похоже на два предыдущих в том смысле, что совпадает со всей строкой и замещает ее той частью, которую требуется сохранить, но так как внутренняя несохраняющая группа при каждом повторении совпадает с единственным словом, механизму регулярных выражений приходится выполнить множество мелких шагов. Проигрыш этого решения в производительности едва ли будет заметен при обработке коротких строк, но на очень длинных строках, содержащих большое число слов, это регулярное выражение может превратиться в «узкое место».

Некоторые реализации механизмов регулярных выражений содержат оптимизации, изменяющие внутренние процессы сопоставления, описанные здесь, вследствие чего эти решения могут показывать производительность чуть выше или чуть ниже, чем мы предположили. Однако простое решение, опирающееся на использование двух регулярных выражений, неизменно показывает приличную производительность на строках любой длины и с любым содержимым, и поэтому оно является лучшим.

См. также

Рецепт 5.13, где объясняется, как заменить повторяющиеся пробельные символы единственным пробелом.

Приемы, использовавшиеся в регулярных выражениях и в тексте замены в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.13 объясняется, как воздействуют на механизм возвратов максимальные и минимальные квантификаторы. В рецепте 2.21 демонстрируется, как вставлять текст, совпавший с сохраняющей группой, в текст замены.

5.13. Замена повторяющихся пробельных символов единственным пробелом

Задача

Продолжая очищать ввод пользователя от лишних символов, требуется заменить повторяющиеся пробельные символы единственным пробе-

лом. Любые символы табуляции, разрыва строки или другие пробельные символы также должны замещаться пробелом.

Решение

Чтобы решить эту задачу, достаточно просто заменить все совпадения со следующими регулярными выражениями единственным пробелом. В рецепте 3.14 показано, как реализовать это решение в программном коде.

Удаление любых пробельных символов

```
\s+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Удаление горизонтальных пробельных символов

```
[ \t\xA0]+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby 1.8

```
[ \t\u00A0]+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python, Ruby 1.9

```
\h+
```

Параметры: нет
Диалекты: PCRE 7.2, Perl 5.10

Обсуждение

Довольно часто в процедуре очистки ввода бывает необходимо заменить повторяющиеся пробельные символы единственным пробелом. Например, при отображении документов HTML повторяющиеся пробельные символы просто игнорируются (с некоторыми исключениями). Удаление повторяющихся пробельных символов может помочь уменьшить размеры некоторых страниц (или хотя бы разделов страниц) без каких-либо отрицательных последствий.

Удаление любых пробельных символов

В этом решении любые последовательности пробельных символов (разрывы строк, символы табуляции, пробелы и прочие) замещаются единственным символом пробела. Поскольку квантификатор ‹+› повторяет символьный класс, совпадающий с любым пробельным символом (‹\s›), один или более раз, даже единственный символ табуляции, например, будет замещен пробелом. Если заменить квантификатор ‹+› на ‹{2,}›, замещаться будут только последовательности из двух или более пробельных символов. Это может привести к уменьшению числа замен и, как

следствие, к увеличению производительности, но в этом случае останутся незамеченными символы табуляции или разрывы строк, которые в противном случае были бы замещены символами пробела. Поэтому какой подход считать лучшим, зависит от желаемого результата.

Удаление горизонтальных пробельных символов

Это решение действует точно так же, как и предыдущее, за исключением того, что оставляет разрывы строк нетронутыми. Замещаются только символы табуляции, пробелы и неразрывные пробелы. На мнемонику HTML, обозначающую неразрывный пробел ( ), это не распространяется.

Диалекты PCRE 7.2 и Perl 5.10 включают сокращение ‹\h› символьного класса, который специально определен для поиска горизонтальных пробельных символов. Он также совпадает с некоторыми необычными горизонтальными пробельными символами.

Применение последовательности ‹\xA0› для поиска совпадений с неразрывными пробелами в Ruby 1.9 может привести к ошибке «invalid multibyte escape» (недопустимое определение многобайтного символа) или к другим ошибкам, связанным с ней, потому что она обозначает символ за пределами диапазона ASCII от ‹\x00› до ‹\x7F›. Поэтому используйте последовательность ‹\u00A0›.

См. также

Рецепт 5.12, где объясняется, как удалить ведущие и завершающие пробельные символы.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

5.14. Экранирование метасимволов регулярных выражений

Задача

Необходимо реализовать возможность использовать в регулярных выражениях строковые литералы, получаемые от пользователя или из других источников. Однако прежде чем встраивать эту строку в регулярное выражение, необходимо экранировать все метасимволы регулярных выражений, чтобы избежать появления непредвиденных эффектов.

Решение

После добавления символа обратного слэша перед каждым символом, который потенциально может иметь специальное назначение в регулярном выражении, можно безопасно использовать получившийся шаблон для поиска литеральной последовательности символов. Все языки программирования, описываемые в этой книге, за исключением JavaScript, имеют встроенные функции или методы, выполняющие эту задачу (перечислены в табл. 5.3). Однако ради полноты обсуждения мы покажем, как решить эту задачу с помощью собственного регулярного выражения даже в языках, имеющих готовые решения.

Встроенные решения

В табл. 5.3 перечислены встроенные функции, предназначенные для решения этой задачи.

Таблица 5.3. Встроенные функции, выполняющие экранирование метасимволов регулярных выражений

Язык программирования	Функция
C#, VB.NET	`Regex.Escape(str)`
Java	`Pattern.quote(str)`
XRegExp	`XRegExp.escape(str)`
Perl	`quotemeta(str)`
PHP	`preg_quote(str, [delimiter])`
Python	`re.escape(str)`
Ruby	`Regexp.escape(str)`

Обратите внимание на отсутствие языка JavaScript (без использования XRegExp) в этом списке – в нем отсутствует встроенная функция, предназначенная для этой цели.

Регулярное выражение

Несмотря на то, что лучше использовать встроенное решение, если оно существует, эту же задачу можно решить с помощью следующего регулярного выражения и замещающего текста (приводится ниже). При этом необходимо выполнить замещение всех совпадений, а не только первого. В рецепте 3.15 приводится программный код, производящий замещение совпадений строкой, которая содержит обратную ссылку. Обратная ссылка необходима, чтобы вернуть в строку совпавший специальный символ с предшествующим ему символом обратного слэша:

```
[[\]{}()*+?.\\|^$\-,&#\s]
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

Следующие строки с замещающим текстом содержат литерал обратного слэша. Строки приводятся без дополнительных символов обратного слэша, которые могут потребоваться для экранирования обратных слэшей при их использовании в строковых литералах в некоторых языках программирования. Подробнее о диалектах замещающего текста рассказывается в рецепте 2.19.

```
\$&
```

Диалекты замещающего текста: .NET, JavaScript

```
\$0
```

Диалекты замещающего текста: .NET, XRegExp

```
\\$&
```

Диалект замещающего текста: Perl

```
\\$0
```

Диалекты замещающего текста: Java, PHP

```
\\\0
```

Диалекты замещающего текста: PHP, Ruby

```
\\\&
```

Диалект замещающего текста: Ruby

```
\\\g<0>
```

Диалект замещающего текста: Python

Пример функции на JavaScript

Ниже приводится пример, демонстрирующий, как использовать регулярное выражение и строку замещающего текста для создания статического метода `RegExp.escape()` в JavaScript:

```
RegExp.escape = function(str) {
    return str.replace(/[[\]{}()*+?.\\|^$\-,&#\s]/g, "\\$&");
};

// Проверка метода...
var str = "<Hello World.>";
var escapedStr = RegExp.escape(str);
alert(escapedStr == "<Hello\\ World\\.>"); // -> true
```

Обсуждение

Регулярное выражение в этом рецепте помещает все метасимволы внутрь единого символьного класса. Рассмотрим каждый из этих символов и разберемся, почему их следует экранировать. Для одних символов потребность их экранировать не так очевидна, как для других:

[{ (

Квадратная скобка ‹[› создает символьный класс. Фигурная скобка ‹{› создает интервальный квантификатор, а также используется в некоторых специальных конструкциях, таких как категории Юникода. Круглые скобки ‹(› и ‹)› используются для группировки, сохранения и в других специальных конструкциях.

* + ?

Эти три символа являются квантификаторами, которые повторяют предшествующий им элемент ноль или более, один или более и ноль или один раз соответственно. Знак вопроса также используется после открывающей круглой скобки для образования специальных видов группировок и других конструкций (то же самое относится и к символу звездочки в Perl 5.10 и PCRE 7).

\ |

Точке соответствует любой символ в испытуемом тексте, обратный слэш экранирует специальные символы или делает литерал символа специальным, а вертикальная черта обеспечивает возможность выбора между несколькими альтернативами.

^ $

Символ крышки и знак доллара являются якорными метасимволами, соответствующими началу или концу текста или строки. Кроме того, символ крышки может использоваться для инвертирования символьного класса.

Остальные символы, соответствующие регулярному выражению, обретают специальное назначение только в специальных случаях. Они включены в список из предосторожности.

]

Закрывающая квадратная скобка завершает символьный класс. Обычно ее не требуется экранировать, но экранировав ее, можно избежать непреднамеренного закрытия символьного класса, если текст, введенный пользователем, вставляется в символьный класс. Следует иметь в виду, что при встраивании текста внутрь символьного класса получившееся регулярное выражение не будет совпадать со встроенной строкой; оно будет совпадать с одним из символов, присутствующих во встраиваемой строке.

-

Дефис образует диапазон внутри символьного класса. Он экранируется здесь, чтобы избежать непреднамеренного создания диапазонов, когда встраиваемый текст оказывается внутри символьного класса.

}

Закрывающая фигурная скобка завершает интервальные квантификаторы и некоторые другие специальные конструкции. В боль-

шинстве диалектов регулярных выражений фигурная скобка интерпретируется как обычный символ, если она не замыкает интервальный квантификатор, поэтому есть вероятность по неосторожности создать интервальный квантификатор там, где его не должно быть, если не экранировать открывающую и закрывающую фигурные скобки при вставке текста в регулярное выражение.

,

Запятая используется внутри интервальных квантификаторов, таких как ‹{1,5}›. Существует вероятность (хотя и небольшая) образовать квантификатор там, где его не было перед вставкой литерального текста в регулярное выражение, если не экранировать запятые.

&

Символ амперсанда включен в этот перечень по той причине, что в диалекте Java два амперсанда, следующие друг за другом, обозначают операцию пересечения символьных классов. В других языках программирования можно безопасно удалить амперсанд из списка символов, которые требуется экранировать, но не будет ошибкой и оставить его.

и пробельные символы

Символ решетки и пробельные символы (соответствующие классу ‹\s›) являются метасимволами, только если включен режим свободного форматирования. Эти символы также не будет ошибкой экранировать в любом случае.

Что касается замещающего текста, одна из пяти конструкций («$&», «\&», «$0», «\0» или «\g<0>») может использоваться для восстановления совпавшего символа вместе с символом обратного слэша. В Perl конструкция $& является переменной, использование которой с любым регулярным выражением приводит к потере его производительности. Если переменная $& используется где-то в другом месте программы на языке Perl, в этом нет ничего особенного, потому что вы уже заплатили полную цену за это. В противном случае лучше будет обернуть все выражение сохраняющей группировкой и вместо $& использовать в замещающем тексте обратную ссылку $1.

Варианты

Как уже объяснялось в разделе «Экранирование блока» в рецепте 2.1, можно экранировать целый блок текста с помощью конструкции ‹\Q···\E›. Возможность экранирования блоков поддерживается только в диалектах Java, PCRE и Perl, но даже в этих языках экранирование блока не обеспечивает защиту от ошибок. Чтобы обеспечить полную безопасность, все равно потребуется экранировать все вхождения \E в строке, которую предполагается внедрить в регулярное выражение. В большинстве случаев гораздо проще использовать описанный выше прием,

не зависящий от используемого языка программирования, основанный на экранировании всех метасимволов регулярных выражений.

См. также

Рецепт 2.1, где обсуждается, как обеспечить совпадение с литералами символов. Однако список символов, обязательных для экранирования, приводимый в этом рецепте, немного короче, потому что в нем не учитываются символы, которые может потребоваться экранировать в режиме свободного форматирования или в случае, когда они могут вставляться произвольно, при работе с длинными шаблонами.

В примере решения на JavaScript в рецепте 5.2 определяется функция, экранирующая любые метасимволы регулярных выражений внутри слова, переданного для поиска. В ней используется сокращенный список специальных символов из рецепта 2.1.

Приемы, использовавшиеся в регулярных выражениях и в тексте замены в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.20 объясняется, как вставлять совпадения с регулярным выражением в замещающий текст.

6

Числа

Регулярные выражения предназначены для работы с текстом и не способны интерпретировать строки из цифр как числа. Для регулярного выражения `56` – это не число пятьдесят шесть, а строка, содержащая два символа, которые отображаются как цифры 5 и 6. Механизм регулярных выражений знает, что это цифры, потому что им соответствует класс `\d` (рецепт 2.3). Но это все, что он знает. Он не знает, что `56` имеет более глубокий смысл, точно так же, как он не знает, что `:-)` – это не просто три знака пунктуации, соответствующие выражению `\p{P}{3}`.

Однако числа – одна из наиболее важных разновидностей вводимой информации, с которыми приходится сталкиваться на практике, и иногда возникает необходимость обрабатывать их с помощью регулярных выражений, вместо того чтобы передавать программному коду, когда требуется ответить на такой вопрос: «Попадает ли число в диапазон от 1 до 100?». Поэтому мы посвятили целую главу вопросам сопоставления регулярных выражений со всеми видами чисел. Сначала мы рассмотрим несколько рецептов, которые могут показаться тривиальными, но на самом деле объясняют важные базовые понятия. В последующих рецептах, где приводятся более сложные регулярные выражения, предполагается, что вы владеете этими основными понятиями.

6.1. Целые числа

Задача

Требуется отыскать различные целые десятичные числа в объемном документе или проверить, представляет ли строковая переменная целое десятичное число.

Решение

Поиск любых положительных целых десятичных чисел в объемном документе:

```
\b[0-9]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли текстовая строка целым положительным десятичным числом:

```
\A[0-9]+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск любых отдельных целых положительных десятичных чисел в объемном документе:

```
(?<=^|\s)[0-9]+(?=$|\s)
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Ruby 1.9

Для Perl и Python необходимо внести изменения в предыдущее решение, потому что они не поддерживают альтернативы с разной длиной совпадений в ретроспективных проверках:

```
(?:^|(?<=\s))[0-9]+(?=$|\s)
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Поиск любых отдельных целых положительных десятичных чисел в объемном документе. При этом допускается попадание ведущих пробельных символов в совпадение:

```
(^|\s)([0-9]+)(?=$|\s)
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск любых целых десятичных чисел, которым может предшествовать знак плюс или минус:

```
[+-]?\b[0-9]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли текстовая строка целым десятичным числом, которому может предшествовать знак плюс или минус:

```
\A[+-]?[0-9]+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[+-]?[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск любого целого десятичного числа с необязательным знаком. При этом допускается наличие пробельного символа между знаком и числом, но в числах без знака ведущие пробелы не допускаются:

```
([+-]**)?\b[0-9]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Целое число – это непрерывная последовательность из одной или более цифр от нуля до девяти. Это легко можно выразить с помощью символьного класса (рецепт 2.3) и квантификатора (рецепт 2.12): ‹[0-9]+›.

Мы предпочитаем явно указывать диапазон ‹[0-9]› и не использовать сокращенную форму записи ‹\d›. В диалектах .NET и Perl метасимвол ‹\d› соответствует любой цифре из любого алфавита, тогда как класс ‹[0-9]› всегда соответствует одной из 10 цифр в таблице ASCII. Если заранее известно, что в испытуемом тексте отсутствуют цифры, не входящие в диапазон ASCII, можно сэкономить несколько нажатий на клавиши и использовать ‹\d› вместо ‹[0-9]›.

Если заранее неизвестно, включает ли текст цифры, не входящие в таблицу ASCII, необходимо задуматься над тем, что делать с найденными совпадениями и что ожидает получить пользователь, чтобы определить, использовать ли метасимвол ‹\d› или символьный класс ‹[0-9]›. Если предполагается преобразовывать текст совпадения с регулярным выражением в целое число, нужно проверить, сможет ли функция, имеющаяся в языке программирования и выполняющая преобразование из строки в число, интерпретировать цифры, не являющиеся символами ASCII. Пользователи, создающие документы с использованием своих родных алфавитов, будут ожидать, что наше программное обеспечение распознает цифры в этих документах.

Кроме того что число является последовательностью цифр, оно также должно стоять обособленно. Например, A4 – это размер бумаги, а не число. Существует несколько способов, позволяющих гарантировать, что регулярное выражение будет совпадать только с числами.

Если необходимо проверить, является ли некоторый текст числом, достаточно окружить регулярное выражение якорными символами, совпадающими с началом и с концом текста. Лучше всего для этого выби-

рать метасимволы ‹\A› и ‹\Z›, потому что их смысл не изменяется. К сожалению, диалект JavaScript их не поддерживает. В JavaScript можно использовать метасимволы ‹^› и ‹$›, но при этом не следует указывать флаг /m, который вынуждает символ крышки и знак доллара совпадать с разрывами строк. В Ruby символ крышки и знак доллара всегда совпадают с разрывами строк, поэтому их использование не может вынудить регулярное выражение совпадать со всем текстом.

Когда поиск чисел производится в пределах объемного текстового документа, простое решение могут дать метасимволы границы слова (рецепт 2.6). Если поместить их до или после элемента регулярного выражения, соответствующего цифре, границы слова обеспечат отсутствие символов слова до или после совпавшей цифры. Например, выражение ‹4› совпадет с цифрой 4 в строке A4. Выражение ‹4\b› тоже обнаружит совпадение, потому что в этой строке после цифры 4 отсутствуют какие-либо символы слова. Выражения ‹\b4› и ‹\b4\b› не обнаружат совпадения в строке A4, потому что проверка ‹\b› будет терпеть неудачу при сопоставлении с позицией между символами A и 4. В регулярных выражениях к символам слова относятся буквы, цифры и знак подчеркивания.

Если в регулярное выражение включается сопоставление с символом, не являющимся символом слова, таким как знак плюс или минус, или пробельный символ, следует быть очень осторожным в использовании границ слова. Чтобы найти совпадение с последовательностью +4 и при этом исключить совпадение с последовательностью +4B, вместо выражения ‹\b\+4\b› следует использовать выражение ‹\+4\b›. Первое выражение не будет совпадать с фрагментом +4, потому что в испытуемой строке перед знаком плюс отсутствует символ слова, чтобы удовлетворить границу слова. Выражение ‹\b\+4\b› обнаружит совпадение +4 в строке 3+4, потому что 3 – это символ слова, а + – нет.

Выражение ‹\+4\b› требует наличия единственной границы слова. В выражении ‹\+\b4\b› первый метасимвол ‹\b› является избыточным. При сопоставлении этого регулярного выражения первый метасимвол ‹\b› всегда будет обнаруживать совпадение с позицией между + и 4 и поэтому никогда ничего не исключит. Первый метасимвол ‹\b› становится необходимым, когда знак плюс является необязательным. Выражение ‹\+?\b4\b› не совпадет с цифрой 4 в тексте A4, а выражение ‹\+?4\b› совпадет.

Границы слова не всегда обеспечивают верное решение. Рассмотрим испытуемый текст $123,456.78. Если попытаться в цикле сопоставить с этой строкой регулярное выражение ‹\b[0-9]+\b›, оно обнаружит совпадения 123, 456 и 78. Знак доллара, запятая и десятичная точка не являются символами слова, поэтому границы слова будут обнаруживать совпадения между цифрами и всеми этими символами. Иногда это именно то, что требуется, а иногда – нет.

Если требуется всего лишь отыскать целые числа, окруженные пробельными символами или началом и концом текста, вместо границ слова следует использовать проверку соседних символов. Проверке ‹(?=$|\s)›

соответствует позиция в самом конце текста или перед пробельным символом (разрывы строк также входят в число пробельных символов). Проверке ‹(?>=^|\s)› соответствует позиция в самом начале текста или после пробельного символа. Метасимвол ‹\s› можно заменить символьным классом, совпадающим с любым желаемым символом, который может появляться перед числом или после него. Порядок работы проверок соседних символов описывается в рецепте 2.16.

Диалекты Perl и Python поддерживают ретроспективные проверки, но они не допускают использования в них альтернатив с разной длиной совпадений. Так как совпадение с ‹^› имеет нулевую длину, а совпадение с ‹\s› включает один символ, альтернативу ‹^› необходимо вынести за пределы ретроспективной проверки. То есть конструкция ‹(?<=^|\s)› в диалектах Perl и Python превращается в конструкцию ‹(?:^|(?<=\s))›. Функционально эти два регулярных выражения идентичны. Просто последнее требует нескольких лишних нажатий на клавиатуре.

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку. Чтобы проверить, находится ли число в начале строки или предшествует ли ему пробельный символ, вместо ретроспективной проверки можно использовать обычную группировку. Недостаток такого подхода состоит в том, что пробельный символ будет включен в совпадение, если число не находится в начале текста. Эту проблему можно легко решить, если часть регулярного выражения, совпадающую с числом, поместить в сохраняющую группу. Пятое регулярное выражение в разделе «Решение» сохраняет пробельный символ в первой сохраняющей группе, а совпадение с целым числом – во второй.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

6.2. Шестнадцатеричные числа

Задача

Требуется отыскать шестнадцатеричные числа в объемном текстовом документе или проверить, является ли значение строковой переменной шестнадцатеричным числом.

Решение

Поиск любых шестнадцатеричных чисел в текстовом документе:

```
\b[0-9A-F]+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b[0-9A-Fa-f]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли строка шестнадцатеричным числом:

```
\A[0-9A-F]+\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[0-9A-F]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск шестнадцатеричных чисел с префиксом `0x`:

```
\b0x[0-9A-F]+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричных чисел с префиксом `&H`:

```
&H[0-9A-F]+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричных чисел, оканчивающихся символом `H`:

```
\b[0-9A-F]+H\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричного значения байта, или 8-битного числа:

```
\b[0-9A-F]{2}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричного значения слова, или 16-битного числа:

```
\b[0-9A-F]{4}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричного значения двойного слова, или 32-битного числа:

```
\b[0-9A-F]{8}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск шестнадцатеричного значения четверного слова, или 64-битного числа:

```
\b[0-9A-F]{16}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск строки шестнадцатеричных значений байтов (то есть четного числа шестнадцатеричных цифр):

```
\b(?:[0-9A-F]{2})+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

При сопоставлении шестнадцатеричных чисел с регулярными выражениями используются те же приемы, что и при сопоставлении целых десятичных чисел. Единственное отличие состоит в том, что символьный класс, соответствующий одной цифре, на этот раз должен включать буквы от A до F. При этом необходимо решить, должны ли эти символы использоваться в верхнем или в нижнем регистре или допускается смешивание регистров символов. Регулярные выражения, рассматриваемые здесь, допускают смешивание регистров символов.

По умолчанию регулярные выражения различают регистр символов. Классу `‹[0-9a-f]›` соответствуют шестнадцатеричные цифры только в нижнем регистре, а классу `‹[0-9A-F]›` – только в верхнем регистре. Чтобы обеспечить возможность совпадения без учета регистра символов, можно использовать класс `‹[0-9a-fA-F]›` или включить режим нечувствительности к регистру символов. Как это сделать в языках программирования, описываемых в этой книге, рассказывается в рецепте 3.4. Первое регулярное выражение в решениях показано дважды – в них используются разные способы обеспечения нечувствительности к регистру символов. Во всех остальных регулярных выражениях используется только второй способ.

Если необходимо, чтобы в шестнадцатеричных числах были допустимы только символы верхнего регистра, можно использовать приведенные регулярные выражения с выключенным режимом нечувствительности к регистру символов. Чтобы обеспечить допустимость только символов нижнего регистра, следует отключить режим нечувствительности к регистру символов и заменить фрагмент `‹A-F›` на `‹a-f›`.

Выражение `‹(?:[0-9A-F]{2})+›` соответствует четному числу шестнадцатеричных цифр. Выражение `‹[0-9A-F]{2}›` соответствует точно двум шест-

надцатеричным цифрам. Выражение ‹(?:[0-9A-F]{2})+› повторяет это сопоставление один или более раз. Применение несохраняющей группировки (рецепт 2.9) здесь совершенно необходимо, потому что квантификатор ‹+› должен повторять комбинацию символьного класса и квантификатора ‹{2}›. Выражение ‹[0-9A-F]{2}+› не будет рассматриваться как ошибочное в диалектах Java, PCRE и Perl 5.10, но оно делает совсем не то, что ожидается. Дополнительный символ ‹+› превращает квантификатор ‹{2}› в захватывающий. В данном случае это не дает никакого эффекта, потому что квантификатор ‹{2}› не может повторить предшествующий ему элемент менее двух раз.

В некоторых решениях показано, как обеспечить совпадения с шестнадцатеричными числами, имеющими префикс или окончание, часто используемые для идентификации шестнадцатеричных чисел. Они используются, чтобы обозначить разницу между десятичными и шестнадцатеричными числами, которые могут состоять исключительно из десятичных цифр. Например, 10 может быть десятичным числом между 9 и 11 или шестнадцатеричным числом между F и 11.

В большинстве решений используются метасимволы границы слова (рецепт 2.6). Границы слова можно использовать при поиске чисел в текстовых документах. Следует отметить, что регулярное выражение, допускающее наличие префикса &H, не имеет метасимвола границы слова в начале. Это обусловлено тем, что амперсанд не является символом слова. Если бы в начале регулярного выражения присутствовал метасимвол границы слова, оно отыскивало бы только шестнадцатеричные числа, следующие сразу же за символом слова.

Если необходимо проверить, является ли весь текст шестнадцатеричным числом, достаточно просто окружить регулярное выражение метасимволами, совпадающими с началом и с концом текста. Лучше всего для этого выбирать метасимволы ‹\A› и ‹\Z›, потому что их смысл не изменяется. К сожалению, диалект JavaScript их не поддерживает. В JavaScript можно использовать метасимволы ‹^› и ‹$›, но при этом не следует указывать флаг /m, который вынуждает символ крышки и знак доллара совпадать с разрывами строк. В Ruby символ крышки и знак доллара всегда совпадают с разрывами строк, поэтому их использование не может вынудить регулярное выражение совпадать со всем текстом.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.3. Двоичные числа

Задача

Требуется отыскать двоичные числа в объемном текстовом документе или проверить, является ли значение строковой переменной двоичным числом.

Решение

Поиск двоичного числа в текстовом документе:

```
\b[01]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли строка двоичным числом:

```
\A[01]+\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[01]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск двоичного числа, начинающегося с префикса `0b`:

```
\b0b[01]+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск двоичного числа, оканчивающегося символом `B`:

```
\b[01]+B\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск двоичного значения байта, или 8-битного числа:

```
\b[01]{8}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск двоичного значения слова, или 16-битного числа:

```
\b[01]{16}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск строки байтов (то есть двоичных цифр, количество которых кратно 8):

```
\b(?:[01]{8})+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Во всех этих регулярных выражениях используются те же приемы, что и в двух предыдущих рецептах. Основное отличие состоит в том, что теперь каждая цифра может быть 0 или 1. Это легко можно обеспечить с помощью символьного класса, включающего всего два символа: ‹[01]›.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.4. Восьмеричные числа

Задача

Требуется отыскать восьмеричные числа в объемном текстовом документе или проверить, является ли значение строковой переменной восьмеричным числом. Восьмеричное число – это число, состоящее из цифр от 0 до 7. Число должно начинаться хотя бы с одного нуля или с префикса 0o.

Решение

Поиск восьмеричного числа в текстовом документе:

```
\b0[0-7]*\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли строка восьмеричным числом:

```
\A0[0-7]*\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^0[0-7]*$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск восьмеричного числа, начинающегося с префикса 0o:

```
\b0o[0-7]+\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Эти регулярные выражения очень напоминают регулярные выражения из предыдущих рецептов в этой главе. Единственное существенное отличие – префикс `0` также является частью самого восьмеричного числа. В частности, отдельная цифра 0 является допустимым восьмеричным числом. Поэтому если в предыдущих рецептах для повторения сопоставления с символьным классом цифр один или более раз использовался квантификатор ‹+›, то в первых двух решениях в этом рецепте используется квантификатор ‹*›, повторяющий попытки сопоставления ноль или более раз. Таким способом обеспечивается возможность совпадения с восьмеричными числами любой длины, включая число 0.

В третьем решении снова использован квантификатор ‹+›. Это обусловлено требованием наличия хотя бы одной цифры после префикса `0o`.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.5. Десятичные числа

Задача

Требуется найти разные типы целых десятичных чисел в объемном текстовом документе или проверить, является ли значение строковой переменной целым десятичным числом. Число не должно начинаться с нуля, так как это отличительная особенность восьмеричных чисел. Но сама цифра «ноль» является допустимым десятичным числом.

Решение

Поиск в текстовом документе любого положительного целого десятичного числа, не начинающегося с нуля:

```
\b(0|[1-9][0-9]*)\b
```

Параметры: нет
Диалекты:.NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли строка положительным целым десятичным числом, не начинающимся с нуля:

```
\A(0|[1-9][0-9]*)\Z
```

Параметры: нет
Диалекты:.NET, Java, PCRE, Perl, Python, Ruby

```
^(0|[1-9][0-9]*)$
```

Параметры: нет
Диалекты:.NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

В рецепте 6.1 приводится множество решений задачи поиска целых десятичных чисел с подробным их описанием. Однако те решения не учитывают, что во многих языках программирования числа, начинающиеся с нуля, интерпретируются как восьмеричные, а не как десятичные. В них просто используется конструкция `‹[0-9]+›`, совпадающая с любыми последовательностями десятичных цифр.

Решения в этом рецепте исключают числа, начинающиеся с нуля, но совпадают с единственным нулем. Вместо конструкции `‹[0-9]+›`, совпадающей с произвольной последовательностью десятичных цифр, в данном рецепте используется конструкция `‹0|[1-9][0-9]*›`, совпадающая либо с единственной цифрой «ноль», либо с десятичным числом, содержащим хотя бы одну цифру и не начинающимся с цифры «ноль». Так как оператор выбора имеет самый низкий приоритет среди всех операторов регулярных выражений, мы использовали группировку, чтобы вывести сопоставление с якорями и границами слов за пределы оператора выбора.

См. также

Рецепт 6.4, где приводятся решения для сопоставления с восьмеричными числами.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.6. Удаление ведущих нулей

Задача

Требуется написать регулярное выражение, совпадающее с целым числом, которое либо возвращает число без ведущих нулей, либо удаляет ведущие нули.

Решение

Регулярное выражение

```
\b0*([1-9][0-9]*|0)\b
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

```
$1
```

Диалекты замещающего текста: .NET, Java, JavaScript, PHP, Perl

```
\1
```

Диалекты замещающего текста: PHP, Python, Ruby

Извлечение чисел в Perl

```
while ($subject =~ m/\b0*([1-9][0-9]*|0)\b/g) {
    push(@list, $1);
}
```

Удаление ведущих нулей в PHP

```
$result = preg_replace('/\b0*([1-9][0-9]*|0)\b/', '$1', $subject);
```

Обсуждение

Для отделения числа от ведущих нулей мы используем сохраняющую группировку. Подвыражение `‹0*›` перед группой совпадает с ведущими нулями, если таковые имеются. Конструкции `‹[1-9][0-9]*›` внутри группы соответствует число, состоящее из одной или более цифр, причем первая цифра числа не может быть нулем. Число может начинаться с нуля, только если это сам ноль. Границы слова гарантируют, что совпадение будет обнаружено с целым числом, а не с его частью, о чем уже говорилось в рецепте 6.1.

Чтобы получить список всех чисел в испытуемом тексте, не содержащих ведущих нулей, необходимо выполнить обход всех совпадений с регулярным выражением, как описывается в рецепте 3.11. Внутри цикла число следует извлекать из первой (и единственной) сохраняющей группы, как описывается в рецепте 3.9. В этом решении показано, как реализовать эту операцию на языке Perl.

Удаление ведущих нулей легко можно выполнить с помощью операции поиска с заменой. Наше регулярное выражение отделяет число от ведущих нулей с помощью сохраняющей группировки. Если заменить все совпадение с регулярным выражением (число вместе с ведущими нулями) текстом совпадения с первой сохраняющей группой, мы фактически удалим ведущие нули. В решении показано, как можно реализовать эту операцию на языке PHP.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.7. Числа в определенном диапазоне

Задача

Требуется обеспечить совпадение с целым числом, попадающим в определенный диапазон. При этом необходимо, чтобы регулярное выражение определяло точный диапазон значений, а не ограничивало количество цифр.

Решение

От 1 до 12 (час или месяц)

```
^(1[0-2]|[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 24 (час):

```
^(2[0-4]|1[0-9]|[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 31 (день месяца):

```
^(3[01]|[12][0-9]|[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 53 (неделя года):

```
^(5[0-3]|[1-4][0-9]|[1-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 59 (минута или секунда):

```
^[1-5]?[0-9]$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 100 (процент):

```
^(100|[1-9]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 100:

```
^(100|[1-9][0-9]?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 32 до 126 (коды отображаемых символов ASCII):

```
^(12[0-6]|1[01][0-9]|[4-9][0-9]|3[2-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 127 (неотрицательное значение байта со знаком):

```
^(12[0-7]|1[01][0-9]|[1-9]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От -128 до 127 (значение байта со знаком):

```
^(12[0-7]|1[01][0-9]|[1-9]?[0-9]|-(12[0-8]|1[01][0-9]|[1-9]?[0-9]))$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 255 (значение байта без знака):

```
^(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 366 (день года):

```
^(36[0-6]|3[0-5][0-9]|[12][0-9]{2}|[1-9][0-9]?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1900 до 2099 (год):

```
^(19|20)[0-9]{2}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 32767 (неотрицательное значение слова со знаком):

```
^(3276[0-7]|327[0-5][0-9]|32[0-6][0-9]{2}|3[01][0-9]{3}|[12][0-9]{4}|↵
```

```
[1-9][0-9]{1,3}|[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От -32768 до 32767 (значение слова со знаком):

```
^(3276[0-7]|327[0-5][0-9]|32[0-6][0-9]{2}|3[01][0-9]{3}|[12][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9]|-(3276[0-8]|327[0-5][0-9]|32[0-6][0-9]{2}|↵
3[01][0-9]{3}|[12][0-9]{4}|[1-9][0-9]{1,3}|[0-9]))$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 65535 (значение слова без знака):

```
^(6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

В предыдущих рецептах выполнялось сопоставление с целыми числами, состоящими из произвольного или определенного количества цифр. Они допускали появление любых цифр в любом разряде числа. Это очень простые регулярные выражения.

Сопоставление с числами в определенном диапазоне, например с числами от 0 до 255, – не такая уж простая задача для регулярных выражений. Нельзя написать выражение ‹[0-255]›. Нет, вы, конечно, можете записать такое выражение, но оно не будет совпадать с числами в диапазоне от 0 до 255. Это будет символьный класс, эквивалентный классу ‹[0125]›, соответствующий единственному символу, который является цифрой 0, 1, 2 или 5.

Так как эти регулярные выражения получились немного длинными, во всех решениях используются якорные метасимволы, чтобы сделать выражения пригодными для проверки, является ли вся строка, например, полученная от пользователя, допустимым числом. В рецепте 6.1 объясняется, как вместо якорных метасимволов можно использовать границы слова и проверки соседних символов в случае применения регулярных выражений для других целей. В обсуждении регулярные выражения будут приводиться без якорных метасимволов и основное внимание будет уделяться работе с диапазонами. Если у вас появится желание использовать одно из этих регулярных выражений, вы должны будете добавить якорные метасимволы или границы слова, чтобы обеспечить совпадение регулярного выражения с полным числом, а не с его частью.

Регулярные выражения сопоставляются с текстом символ за символом. Если необходимо получить совпадение с числом, состоящим более чем

из одной цифры, придется предусмотреть все возможные комбинации цифр. Основными стандартными строительными блоками таких выражений являются символьные классы (рецепт 2.3) и конструкции выбора (рецепт 2.8).

В символьных классах можно использовать диапазоны для одной цифры, такие как ‹[0-5]›. Это возможно благодаря тому, что цифры от 0 до 9 занимают непрерывную область в таблицах символов ASCII и Юникод. Классу ‹[0-5]› соответствует один из шести символов, точно так же как классам ‹[j-o]› и ‹[\x09-\x0E]› соответствуют другие диапазоны из шести символов.

Когда число из определенного диапазона присутствует в тексте, оно занимает некоторое количество следующих друг за другом позиций. Каждую позицию занимает некоторая цифра из определенного диапазона. Числа из одних диапазонов, например от 12 до 24, занимают фиксированное количество позиций. Числа из других диапазонов, например от 1 до 12, могут занимать переменное количество позиций. Диапазон цифр, допустимых в каждой позиции, может зависеть или не зависеть от цифр, стоящих в других позициях. В диапазоне чисел от 40 до 59 позиции не зависят друг от друга. В диапазоне чисел от 44 до 55 позиции уже являются взаимозависимыми.

Проще всего представить те диапазоны, числа из которых занимают фиксированное количество независимых позиций, такие как от 40 до 59. Чтобы записать такие числа в виде регулярного выражения, достаточно просто объединить несколько символьных классов. При этом нужно использовать один символьный класс для каждой позиции, указывая в них допустимые диапазоны цифр в той или иной позиции.

```
[45][0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Для представления чисел из диапазона от 40 до 59 требуются две цифры. Поэтому нам потребовались два символьных класса. Первая цифра может быть 4 или 5. Символьный класс ‹[45]› соответствует любой из этих цифр. Вторая цифра может быть любой из 10 цифр. Совпадение с этой цифрой обеспечивается классом ‹[0-9]›.

Вместо класса ‹[0-9]› можно было бы использовать его сокращенную форму записи ‹\d›. Мы используем явный диапазон ‹[0-9]›, чтобы сохранить единство стиля записи символьных классов и поддержать удобочитаемость. Уменьшение числа символов обратного слэша в регулярных выражениях также очень полезно при работе с такими языками программирования, как Java, которые требуют экранировать обратные слэши в строковых литералах.

Числа в диапазоне от 44 до 55 также занимают две позиции, но они не являются взаимонезависимыми. Первая цифра может быть 4 или 5. Ес-

ли первая цифра 4, вторая цифра должна попадать в диапазон от 4 до 9, охватывая числа от 44 до 49. Если первая 5, вторая цифра должна попадать в диапазон от 0 до 5, охватывая числа от 50 до 55. В регулярном выражении можно просто объединить эти два диапазона чисел с помощью конструкции выбора:

```
4[4-9]|5[0-5]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

С помощью конструкции выбора мы сообщили механизму регулярных выражений, что требуется найти совпадение с ‹4[4-9]› или ‹5[0-5]›. Среди всех операций регулярных выражений оператор выбора имеет наименьший приоритет, благодаря чему не требуется группировать цифры, как в выражении ‹(4[4-9])|(5[0-5)›.

При необходимости в конструкции выбора можно объединить столько диапазонов, сколько потребуется. Числа в диапазоне от 34 до 65 также занимают две взаимозависимые позиции. Первая цифра может быть любой цифрой в диапазоне от 3 до 6. Если первая цифра 3, вторая должна попадать в диапазон от 4 до 9. Если первая цифра 4 или 5, вторая цифра может быть любой. Если первая цифра 6, вторая должна попадать в диапазон от 0 до 5:

```
3[4-9]|[45][0-9]|6[0-5]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точно так же, как мы использовали конструкцию выбора, чтобы разбить диапазон с взаимозависимыми позициями на несколько диапазонов с взаимонезависимыми позициями, мы можем использовать конструкцию выбора, чтобы разбить диапазон чисел с переменным числом позиций на несколько диапазонов с фиксированным числом позиций. Числа в диапазоне от 1 до 12 могут занимать одну или две позиции. Выделим из него диапазон от 1 до 9, занимающий одну позицию, и диапазон от 10 до 12, занимающий две позиции. Позиции в каждом из этих двух диапазонов являются взаимонезависимыми, поэтому нам не потребуется разбивать их в свою очередь:

```
1[0-2]|[1-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Мы поместили диапазон чисел с двумя цифрами перед диапазоном с одной цифрой. Это было сделано преднамеренно, потому что механизм регулярных выражений нетерпелив по своей природе. Он выполняет просмотр выражения слева направо и останавливается, как только обнаруживает первое же совпадение. Если представить, что имеется испытуемый текст 12, выражение ‹1[0-2]|[1-9]› обнаружит совпадение 12, тогда как выражение ‹[1-9]|1[0-2]› совпадет только с цифрой ‹1›. Первой будет

опробована альтернатива ‹[1-9]›. Поскольку она благополучно совпадет с 1, механизм регулярных выражений не будет даже пытаться проверить альтернативу ‹1[0-2]›, которая может обеспечить «лучшее» решение.

Некоторые механизмы регулярных выражений не являются нетерпеливыми

POSIX-совместимые механизмы регулярных выражений и механизмы ДКА не следуют этому правилу. Они опробуют все альтернативы и возвращают самое длинное совпадение. Однако все диалекты, рассматриваемые в этой книге, являются механизмами НКА, которые не выполняют дополнительную работу, требуемую стандартом POSIX. Все они сообщат, что выражение ‹[1-9]|1[0-2]› совпадает с 1 в тексте 12.

На практике список альтернатив обычно окружается якорными метасимволами или границами слова. В этом случае порядок следования альтернатив не имеет никакого значения. Оба выражения, ‹^([1-9]|1[0-2])$› и ‹^(1[0-2]|[1-9])$›, в любом из диалектов, описываемых в этой книге, обнаружат совпадение с 12 в тексте 12, так же как POSIX-совместимые механизмы регулярных выражений и механизмы ДКА. Якорные метасимволы требуют, чтобы регулярное выражение совпало либо со всей строкой целиком, либо не совпало вообще. Определения механизмов ДКА и НКА приводятся во врезке «История появления термина «регулярное выражение» в главе 1.

Диапазон от 85 до 117 включает числа, которые могут иметь две разные длины. Числа в диапазоне от 85 до 99 занимают две позиции, а числа в диапазоне от 100 до 117 – три. Позиции в этих диапазонах являются взаимозависимыми, поэтому необходимо произвести дополнительное деление. В случае диапазона, занимающего две позиции: если первая цифра 8, вторая должна попадать в диапазон от 5 до 9. Если первая цифра 9, вторая может быть любой цифрой. В случае диапазона, занимающего три позиции: первая цифра может быть только 1. Если во второй позиции находится цифра 0, тогда в третьей позиции может находиться любая цифра. Но если вторая цифра 1, тогда третья должна попадать в диапазон от 0 до 7. В результате мы получаем четыре диапазона: от 85 до 89, от 90 до 99, от 100 до 109 и от 110 до 117. Несмотря на то что количество диапазонов увеличилось, регулярное выражение остается таким же прямолинейным, как и предыдущие:

```
8[5-9]|9[0-9]|10[0-9]|11[0-7]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы обеспечить совпадение регулярного выражения с диапазонами чисел, необходимо просто выполнять деление на диапазоны, пока все диапазоны не будут иметь фиксированное число взаимонезависимых позиций. Если следовать этому правилу, всегда будут получаться корректные регулярные выражения, которые легко читать и сопровождать, даже если они будут иметь внушительную длину.

Существует несколько приемов, которые позволят уменьшить длину регулярных выражений. Например, регулярное выражение, совпадающее с числами из диапазона от 0 до 65535, с помощью предыдущего правила можно записать так:

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9][0-9]|6[0-4][0-9][0-9][0-9]|↵
[1-5][0-9][0-9][0-9][0-9]|[1-9][0-9][0-9][0-9]|[1-9][0-9][0-9]|↵
[1-9][0-9]|[0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Это регулярное выражение прекрасно справляется со своей задачей, и невозможно придумать регулярное выражение, которое выполнялось бы быстрее. Любые оптимизации, какие можно было бы применить (например, в выражении присутствует несколько альтернатив, начинающихся с 6), будут выполнены механизмом регулярных выражений в процессе компиляции выражения. Нет никакой необходимости тратить свое время впустую, чтобы усложнить выражение в надежде сделать его быстрее. Однако есть возможность сделать регулярное выражение короче, чтобы сэкономить время на вводе с клавиатуры и при этом сохранить удобочитаемость.

В некоторых альтернативах присутствуют идентичные символьные классы, следующие друг за другом. Существует возможность убрать дубликаты, воспользовавшись квантификаторами. Полная информация о квантификаторах приводится в рецепте 2.12.

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↵
[1-9][0-9]{3}|[1-9][0-9]{2}|[1-9][0-9]|[0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Часть ‹[1-9][0-9]{3}|[1-9][0-9]{2}|[1-9][0-9]› регулярного выражения содержит три очень похожих альтернативы и во всех них присутствует одна и та же пара символьных классов. Единственное различие заключается в количестве повторений второго класса. Мы легко можем объединить их в конструкцию ‹[1-9][0-9]{1,3}›.

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Любые последующие ухищрения только ухудшат удобочитаемость. Например, можно было бы отделить ведущую цифру 6 от первых четырех альтернатив:

```
6(?:553[0-5]|55[0-2][0-9]|5[0-4][0-9]{2}|[0-4][0-9]{3})|[1-5][0-9]{4}|↵
[1-9][0-9]{1,3}|[0-9]
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Но в действительности это регулярное выражение получилось на один символ длиннее, потому что в него пришлось добавить несохраняющую группировку, чтобы отделить цифру 6 от других альтернатив. Этот прием не дает преимуществ в скорости ни в одном из диалектов регулярных выражений, рассматриваемых в этой книге. Все они выполняют эту оптимизацию внутренними средствами.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений. В рецепте 6.8 демонстрируется, как реализовать проверку шестнадцатеричных чисел в определенном диапазоне.

В рецепте 4.12 демонстрируется, как исключать определенные числа из диапазона допустимых чисел, используя негативную опережающую проверку.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.8. Шестнадцатеричные числа в определенном диапазоне

Задача

Требуется обеспечить совпадение с шестнадцатеричным числом, попадающим в определенный диапазон. При этом необходимо, чтобы регулярное выражение определяло точный диапазон значений, а не ограничивало количество цифр.

Решение

От 1 до C (от 1 до 12: час или месяц):

```
^[1-9a-c]$
```

Параметры: нечувствительность к регистру символов

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 18 (от 1 до 24: час):

```
^(1[0-8]|[1-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 1F (от 1 до 31: день месяца):

```
^(1[0-9a-f]|[1-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 35 (от 1 до 53: неделя года):

```
^(3[0-5]|[12][0-9a-f]|[1-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 3B (от 0 до 59: минута или секунда):

```
^(3[0-9a-b]|[12]?[0-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 64 (от 0 до 100: процент):

```
^(6[0-4]|[1-5]?[0-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 64 (от 1 до 100):

```
^(6[0-4]|[1-5][0-9a-f]|[1-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 20 до 7E (от 32 до 126: коды отображаемых символов ASCII):

```
^(7[0-9a-e]|[2-6][0-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 7F (от 0 до 127: 7-битное число):

```
^[1-7]?[0-9a-f]$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до FF (от 0 до 255: 8-битное число):

```
^[1-9a-f]?[0-9a-f]$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 1 до 16E (от 1 до 366: день года):

```
^(16[0-9a-e]|1[0-5][0-9a-f]|[1-9a-f][0-9a-f]?)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 76C до 833 (от 1900 до 2099: год):

```
^(83[0-3]|8[0-2][0-9a-f]|7[7-9a-f][0-9a-f]|76[c-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до 7FFF: (от 0 до 32767: 15-битное число):

```
^([1-7][0-9a-f]{3}|[1-9a-f][0-9a-f]{1,2}|[0-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

От 0 до FFFF: (от 0 до 65535: 16-битное число):

```
^([1-9a-f][0-9a-f]{1,3}|[0-9a-f])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Между сопоставлением с диапазонами десятичных чисел и шестнадцатеричных чисел нет никакой разницы. Как уже говорилось в предыдущем разделе, необходимо просто выполнять деление на диапазоны, пока каждый из получившихся диапазонов не будет иметь фиксированное количество взаимонезависимых позиций. После это останется только выбрать символьный класс для каждой позиции и объединить диапазоны с помощью конструкции выбора.

Так как буквы и цифры занимают отдельные области в таблицах символов ASCII и Юникода, невозможно использовать такой класс, как `‹[0-F]›`, для сопоставления с любой из шестнадцатеричных и десятичных цифр. Этот символьный класс будет совпадать с шестнадцатеричными и десятичными цифрами, но он также будет совпадать со знаками пунктуации, расположенными в таблице ASCII между цифрами и буквами. Поэтому следует использовать символьный класс с двумя диапазонами: `‹[0-9A-F]›`.

Другая проблема связана с чувствительностью к регистру символов. По умолчанию регулярные выражения различают символы разных регистров. Классу `‹[0-9A-F]›` соответствуют только символы верхнего регистра, а классу `‹[0-9a-f]›` — только нижнего. Класс `‹[0-9A-Fa-f]›` обеспечивает совпадение с символами любого регистра.

Явно указывать диапазоны символов верхнего и нижнего регистров в каждом классе может быстро стать утомительным занятием. Гораздо проще включить режим нечувствительности к регистру символов. Как это сделать в том или ином языке программирования, рассказывается в рецепте 3.4.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.9. Целые числа с разделителями

Задача

Требуется отыскать различные целые десятичные числа в объемном документе или проверить, представляет ли строковая переменная целое число. Для удобочитаемости группы разрядов в числах могут разделяться символами подчеркивания. Числа не могут начинаться или заканчиваться символом подчеркивания. Числа могут быть десятичными, восьмеричными, шестнадцатеричными и двоичными. Шестнадцатеричные и двоичные числа могут начинаться с префикса `0x` и `0b`, соответственно.

Примерами допустимых чисел являются следующие последовательности: `0b0111_1111_1111_1111_1111_1111_1111_1111`, `0177_7777_7777`, `2_147_483_647` и `0x7fff_ffff`.

Решение

Поиск в объемном документе любого десятичного или восьмеричного целого числа с необязательными символами подчеркивания:

```
\b[0-9]+(_+[0-9]+)*\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск в объемном документе любого шестнадцатеричного целого числа с необязательными символами подчеркивания:

```
\b0x[0-9A-F]+(_+[0-9A-F]+)*\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск в объемном документе любого двоичного целого числа с необязательными символами подчеркивания:

```
\b0b[01]+(_+[01]+)*\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск в объемном документе любого двоичного, восьмеричного, десятичного или шестнадцатеричного целого числа с необязательными символами подчеркивания:

```
\b([0-9]+(_+[0-9]+)*|0x[0-9A-F]+(_+[0-9A-F]+)*|0b[01]+(_+[01]+)*)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверка, является ли текстовая строка двоичным, восьмеричным, десятичным или шестнадцатеричным целым числом с необязательными символами подчеркивания:

```
\A([0-9]+(_+[0-9]+)*|0x[0-9A-F]+(_+[0-9A-F]+)*|0b[01]+(_+[01]+)*)\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([0-9]+(_+[0-9]+)*|0x[0-9A-F]+(_+[0-9A-F]+)*|0b[01]+(_+[01]+)*)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Подробное описание особенностей сопоставления с целыми числами приводится в рецептах 6.1, 6.2 и 6.3. В этих рецептах не допускается присутствие символа подчеркивания в числах. В регулярных выражениях, которые приводятся в них, легко можно обойтись конструкциями `‹[0-9]+›`, `‹[0-9A-F]+›` и `‹[01]+›` для поиска совпадений с десятичными, шестнадцатеричными и двоичными числами.

Если необходимо интерпретировать символы подчеркивания как допустимые, достаточно просто добавить подчеркивание в эти три символьных класса. Но по условиям задачи числа не могут начинаться или заканчиваться символами подчеркивания. Первый и последний символ в числах должен быть цифрой. Кому-то может показаться, что эта проблема легко решается с помощью выражения `‹[0-9][0-9_]+[0-9]›`. Но это не так – оно будет терпеть неудачу на числах, состоящих из единственной цифры. Поэтому приходится использовать более сложное решение.

В решении `‹[0-9]+(_+[0-9]+)*›` подвыражение `‹[0-9]+›` обеспечивает совпадение с начальной цифрой или цифрами, как и прежде. Далее следует конструкция `‹(_+[0-9]+)*›`, допускающая появление символов подчеркивания при условии, что за подчеркиваниями следует одна или более цифр. Подвыражение `‹_+›` разрешает появление любого количества сим-

волов подчеркивания, следующих подряд, а подвыражение ‹[0-9]+› – присутствие любого количества цифр после подчеркиваний. Эти два подвыражения заключены в группу, повторяемую ноль или более раз квантификатором ‹*›. Это обеспечивает возможность появления любого количества символов подчеркивания с цифрами между ними и после них, при этом допускается также совпадение с числами без символов подчеркивания в них.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.10. Вещественные числа

Задача

Требуется обеспечить совпадение с вещественным числом и иметь возможность определять, являются ли знак, целая, дробная и экспоненциальная части числа обязательными, необязательными или недопустимыми. Здесь не требуется, чтобы регулярное выражение ограничивалось определенным диапазоном значений, потому что эту проверку лучше реализовать в программном коде, как описывалось в рецепте 3.12.

Решение

Обязательны знак, целая, дробная и экспоненциальная части.

```
^[-+][0-9]+\.[0-9]+[eE][-+]?[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обязательны знак, целая и дробная части, экспоненциальная часть недопустима.

```
^[-+][0-9]+\.[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Знак необязателен, целая и дробная части обязательны, экспоненциальная часть недопустима.

```
^[-+]?[0-9]+\.[0-9]+$
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак и целая часть, дробная часть обязательна, экспоненциальная часть недопустима.

```
^[-+]?[0-9]*\.[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак, целая и дробная части. Если целая часть опущена, дробная часть становится обязательной. Если опущена дробная часть, десятичная точка также должна быть опущена. Экспоненциальная часть недопустима.

```
^[-+]?([0-9]+(\.[0-9]+)?|\.[0-9]+)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак, целая и дробная части. Если целая часть опущена, дробная часть становится обязательной. Если опущена дробная часть, десятичная точка становится необязательной. Экспоненциальная часть недопустима.

```
^[-+]?([0-9]+(\.[0-9]*)?|\.[0-9]+)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак, целая и дробная части. Если целая часть опущена, дробная часть становится обязательной. Если опущена дробная часть, десятичная точка также должна быть опущена. Экспоненциальная часть необязательна.

```
^[-+]?([0-9]+(\.[0-9]+)?|\.[0-9]+)([eE][-+]?[0-9]+)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательны знак, целая и дробная части. Если целая часть опущена, дробная часть становится обязательной. Если опущена дробная часть, десятичная точка становится необязательной. Экспоненциальная часть необязательна.

```
^[-+]?([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][-+]?[0-9]+)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Предыдущее регулярное выражение, исправленное для поиска числа в текстовом документе:

```
[-+]?(\b[0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][-+]?[0-9]+\b)?
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Все регулярные выражения заключаются в якорные метасимволы (рецепт 2.5), чтобы обеспечить проверку, является ли весь испытуемый текст вещественным числом, в противовес поиску вещественного числа в текстовом документе. Если потребуется выполнить поиск вещественного числа в объемном тексте, можно использовать границы слова или проверки соседних символов, как описывается в рецепте 6.1.

Решения, не допускающие наличия необязательных частей, достаточно прямолинейны – они просто перечисляют требуемые соответствия слева направо. Символьные классы (рецепт 2.3) совпадают со знаком, цифрами и символом e. Квантификаторы ‹+› и ‹?› (рецепт 2.12) допускают появление любого количества цифр и необязательный знак экспоненты.

Сделать необязательными знак и целую часть числа достаточно просто. Знак вопроса, следующий за символьным классом с символами знака, делает его необязательным. Замена квантификатора ‹+› на ‹*› обеспечит возможность повторения цифр целой части числа ноль или более раз вместо одного или более раз.

Сложности начинают возникать, когда необходимо сделать необязательными знак, целую и дробную части числа. Хотя все они сами по себе и являются необязательными, тем не менее они не могут быть необязательными одновременно, так как пустая строка не является допустимым вещественным числом. Простое решение ‹[-+]?[0-9]*\.?[0-9]*› будет совпадать со всеми допустимыми вещественными числами, но оно также будет совпадать и с пустой строкой, а так как мы опустили якорные метасимволы, это регулярное выражение будет совпадать с пустыми строками между любыми двумя символами в испытуемом тексте. Если это регулярное выражение применить в операции поиска с заменой к тексту 123abc456, используя замещающий текст «{$&}», в результате будет получена строка {123}{}a{}b{}c{456}{}. Регулярное выражение корректно совпадет с фрагментами 123 и 456, но оно также обнаружит совпадения нулевой длины во всех остальных попытках.

Создавая регулярное выражение, когда каждый элемент сам по себе может быть необязательным, очень важно учесть, могут ли другие элементы оставаться необязательными, если один из них фактически будет опущен. Вещественное число должно содержать хотя бы одну цифру.

В решениях к этому рецепту подробно поясняется, что когда целая или дробная часть является необязательной, тем не менее требуется присутствие какой-либо из них. Кроме того, отдельно поясняется, какие решения интерпретируют последовательность 123. как вещественное число, а какие – как целое число, за которым следует точка, не являющаяся частью этого числа. Например, в языках программирования завершающая точка может рассматриваться как оператор конкатенации или как первая точка в операторе диапазона, состоящем из двух точек.

Чтобы удовлетворить требование, не допускающее ситуации, когда целая и дробная части могут быть опущены одновременно, для реализации двух ситуаций мы использовали конструкцию выбора (рецепт 2.8) внутри группировки (рецепт 2.9). Выражение ‹[0-9]+(\.[0-9]+)?› соответствует ситуации с обязательной целой частью и необязательной дробной. Выражение ‹\.[0-9]+› соответствует одной дробной части.

Объединение ‹[0-9]+(\.[0-9]+)?|\.[0-9]+› охватывает все три возможные ситуации. Первая альтернатива совпадает с числами, содержащими целую и дробную части, а также с числами без дробной части. Вторая альтернатива совпадает только с дробной частью. Поскольку оператор выбора имеет самый низкий приоритет, мы заключили обе альтернативы в группу, после чего получили возможность добавлять эту конструкцию в более длинные регулярные выражения.

Выражение ‹[0-9]+(\.[0-9]+)?|\.[0-9]+› требует, чтобы десятичная точка была опущена в случае отсутствия дробной части. Если точка может присутствовать в числе даже в отсутствие дробной части, можно использовать выражение ‹[0-9]+(\.[0-9]*)?|\.[0-9]+›. Первая альтернатива в этом регулярном выражении по-прежнему объединена с квантификатором ‹?›, который обеспечивает ее необязательность. Различие заключается в том, что теперь сами цифры дробной части объявлены необязательными. Мы заменили квантификатор ‹+› (один или более) на ‹*› (ноль или более). В результате первая альтернатива этого регулярного выражения совпадает с целой и необязательной дробной частью, причем дробная часть может быть представлена десятичной точкой с цифрами или одной десятичной точкой. Вторая альтернатива осталась без изменений.

Этот последний пример интересен тем, что изменение одного из требований привело к изменению квантификатора в регулярном выражении. Изменилось требование к точке, которая теперь стала необязательной сама по себе, а не в комбинации с цифрами дробной части. Чтобы удовлетворить изменившемуся требованию, мы поменяли квантификатор у символьного класса, соответствующего цифрам дробной части. Такое решение стало возможным благодаря тому, что точка и символьный класс уже были сгруппированы, что делает их необязательными одновременно.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.11. Числа с разделителями групп разрядов

Задача

Требуется обеспечить совпадение с числами, в которых запятая используется для разделения групп разрядов, а точка – для отделения дробной части.

Решение

Обязательные целая и дробная части:

```
^[0-9]{1,3}(,[0-9]{3})*\.[0-9]+$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обязательная целая часть и необязательная дробная. Если дробная часть опущена, десятичная точка также должна быть опущена.

```
^[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Необязательная целая часть и необязательная дробная. Если дробная часть опущена, десятичная точка также должна быть опущена.

```
^([0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?|\.[0-9]+)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Предыдущее регулярное выражение, исправленное для поиска числа в текстовом документе:

```
\b[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?\b|\.[0-9]+\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Поскольку все эти регулярные выражения предназначены для того, чтобы обеспечить совпадение с вещественными числами, в них применяются те же приемы, что и в предыдущем рецепте. Единственное отличие состоит в том, что теперь для сопоставления с целой частью вместо выражения `‹[0-9]+›` используется выражение `‹[0-9]{1,3}(,[0-9]{3})*›`. Это регулярное выражение совпадает с последовательностью от 1 до 3 цифр, за которой следует ноль или более групп, состоящих из запятой и трех цифр.

Мы не можем использовать выражение `‹[0-9]{0,3}(,[0-9]{3})*›`, чтобы обеспечить необязательность целой части, потому что оно совпадает с числами, начинающимися с запятой (например, ,123). Это та же ловушка со всеми необязательными элементами, которая описывалась в предыду-

щем рецепте. Чтобы сделать целую часть необязательной, мы не изменяем элемент регулярного выражения, соответствующий целой части числа, а делаем необязательной всю целую часть целиком. Последние два регулярных выражения в решении делают это с помощью конструкции выбора. Регулярное выражение для необязательной целой части и обязательной дробной объединено с регулярным выражением, совпадающим с дробной частью при отсутствующей целой части. В результате было получено регулярное выражение, в котором и целая, и дробная части являются необязательными, но не одновременно.

См. также

Все рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений. В рецепте 6.12 демонстрируется, как реализовать добавление разделителей групп разрядов в числа, где они отсутствуют.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

6.12. Добавление в числа разделителей групп разрядов

Задача

Требуется добавить запятые, играющие роль разделителей групп разрядов, в числа из четырех и более цифр. Эту операцию может потребоваться выполнять и с отдельными числами, и со всеми числами в строке или в файле.

Например, следующий текст:

```
There are more than 7000000000 people in the world today.
```

необходимо преобразовать так:

```
There are more than 7,000,000,000 people in the world today.
```

Не во всех странах и языках в качестве разделителя групп разрядов используется один и тот же символ. В решениях, представленных здесь, используется запятая, но кто-то с той же целью использует точки, кто-то символы подчеркивания, кто-то апострофы или пробелы. При необходимости вы можете заменить запятые в предлагаемых вариантах текста замены на более подходящие для вас символы.

Решение

Следующие решения одинаково хорошо справляются и с отдельными числами, и со всеми числами в строке. Они создавались для использования в операциях поиска с заменой всех совпадений.

Простое решение

Регулярное выражение:

```
[0-9](?=(?:[0-9]{3})+(?![0-9]))
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Данное регулярное выражение с успехом можно использовать во всех диалектах, охватываемых этой книгой, а вот замещающий текст оказывается менее переносимым.

Замещающий текст:

```
$&,
```

Диалекты замещающего текста: .NET, JavaScript, Perl

```
$0,
```

Диалекты замещающего текста: .NET, Java, XRegExp, PHP

```
\0,
```

Диалекты замещающего текста: PHP, Ruby

```
\&,
```

Диалект замещающего текста: Ruby

```
\g<0>,
```

Диалект замещающего текста: Python

Эти строки замещающего текста вставляют обратно совпавшее число, используя обратную ссылку (в данном случае целым совпадением будет единственная цифра), и запятую. Это регулярное выражение с замещающим текстом можно использовать в программной реализации операции поиска с заменой, как описывается в рецепте 3.15.

Определение позиции вставки разделителя с помощью ретроспективной проверки

Регулярное выражение:

```
(?<=[0-9])(?=(?:[0-9]{3})+(?![0-9]))
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Замещающий текст:

```
,
```

Диалекты замещающего текста: .NET, Java, Perl, PHP, Python, Ruby

В рецепте 3.14 объясняется, как использовать это регулярное выражение в программной реализации операции поиска с заменой.

Эта версия не может использоваться в JavaScript и Ruby 1.8, потому что эти диалекты не поддерживают ретроспективную проверку. Зато на этот раз замещающий текст оказался общим для всех диалектов, потому что он состоит из единственной запятой без каких-либо обратных ссылок.

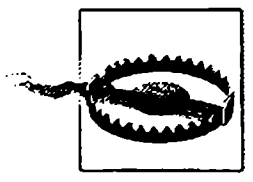
Обсуждение

Введение

Добавление разделителей групп разрядов в числа внутри документа, в данные и в выводе программы – это простой и эффективный путь повышения удобочитаемости.

В некоторых языках программирования, описываемых в этой книге, имеются встроенные методы для добавления в числа разделителей групп разрядов в зависимости от национальных настроек. Например, в языке Python можно использовать инструкцию `locale.format('%d', 1000000, True)`, чтобы преобразовать числа `1000000` в строку `'1,000,000'`, если предположить, что в национальных настройках в качестве разделителя групп разрядов определена запятая. С другими национальными настройками роль такого разделителя может играть точка, символ подчеркивания, апостроф или пробел.

Однако национальные настройки не всегда доступны, надежны или соответствуют поставленной задаче. В мире финансов, например, для разделения групп разрядов принято использовать запятую независимо от национальных настроек. При работе в текстовом редакторе проблема интернационализации вообще может оказаться неразрешимой. По этим причинам и для простоты решения в этом рецепте предполагается, что в качестве разделителей групп разрядов всегда используется запятая. В разделе «Варианты», следующем ниже, дополнительно предполагается, что роль разделителя целой и дробной части всегда играет точка. Если вам потребуется использовать иные символы, просто подставьте их.



Добавление разделителей групп разрядов во все числа в файле или строке может улучшить удобочитаемость, однако прежде чем делать это, следует понять, какого рода содержимое находится в этой строке или файле. Например, едва ли будет правильным добавлять разделители в числовые идентификаторы, даты и почтовые индексы. Документы и данные, содержащие подобные числовые обозначения, являются не лучшими кандидатами на автоматическую обработку процедурой добавления разделителей групп разрядов.

Простое решение

Этому регулярному выражению соответствует любая одна цифра, правее которой находятся другие цифры, число которых кратно трем. Оно найдет два совпадения в строке 12345678 – цифры 2 и 5. Для всех остальных цифр не выполняется условие кратности трем количества цифр, следующих за ними.

Используя обратную ссылку с индексом 0, соответствующий замещающий текст вставляет совпавшую цифру обратно, добавляя вслед за ней запятую. В результате получается строка 12,345,678. Все просто!

Чтобы объяснить, как регулярное выражение обнаруживает совпадение с цифрой, необходимо разбить его на две части. Первая часть – это начальный символьный класс ‹[0-9]›, которому соответствует любая одиночная цифра. Вторая часть – позитивная опережающая проверка ‹(?=(?:[0-9]{3})+(?![0-9]))›, которая вызывает отказ, если число цифр, следующих далее, не кратно трем. Иными словами, опережающая проверка гарантирует совпадение регулярного выражения только с теми цифрами, за которыми должна следовать запятая. Подробнее о том, как действует опережающая проверка, рассказывается в рецепте 2.16.

Выражению внутри опережающей проверки ‹(?:[0-9]{3})+› соответствуют группы из трех цифр. Негативная опережающая проверка ‹(?![0-9])›, следующая далее, гарантирует отсутствие дополнительных цифр, следующих за последней группой из трех цифр. В противном случае внешняя позитивная опережающая проверка удовлетворялась бы любым количеством последующих цифр, если оно не меньше трех.

Определение позиции вставки разделителя с помощью ретроспективной проверки

Это регулярное выражение является адаптацией предыдущего, но ему вообще не соответствует никакая цифра – оно совпадает только с позициями в числах, куда следует вставить запятую. Это позиции, где слева и справа есть цифры, причем справа число цифр кратно трем, а слева должна находиться хотя бы одна цифра.

Опережающая проверка используется для поиска групп из трех цифр справа, как и в предыдущем регулярном выражении. Однако это регулярное выражение начинается не с символьного класса ‹[0-9]›, совпадающего с единственной цифрой, а с проверки наличия слева хотя бы одной цифры с помощью позитивной ретроспективной проверки ‹(?<=[0-9])›. Без ретроспективной проверки регулярное выражение совпало бы, например, с позицией слева от числа 123, в результате чего алгоритм поиска с заменой преобразовал бы его в ,123. Подробнее о том, как действуют ретроспективные и опережающие проверки, рассказывается в рецепте 2.16.

JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку, поэтому в этих диалектах невозможно использовать данное регулярное выражение.

Варианты

Предотвращение добавления запятых после десятичной точки

Предыдущие регулярные выражения добавляют запятые в любые последовательности из четырех и более цифр. Самая большая проблема этого простого решения состоит в том, что оно может добавлять запятые после цифр, составляющих дробную часть числа, если за десятичной точкой следует хотя бы четыре цифры. Исправить этот недостаток можно двумя способами, описанными ниже.

С помощью бесконечной ретроспективной проверки. Проблему можно легко решить, если внутри ретроспективной проверки использовать бесконечный квантификатор, такой как ‹+›, или хотя бы квантификатор с достаточно большим количеством повторений, такой как ‹{1,100}›.

Регулярное выражение:

```
[0-9](?=(?:[0-9]{3})+(?![0-9]))(?<!\.[0-9]+)
```

Параметры: нет
Диалект: .NET

```
[0-9](?=(?:[0-9]{3})+(?![0-9]))(?<!\.[0-9]{1,100})
```

Параметры: нет
Диалекты: .NET, Java

Замещающий текст:

```
$0,
```

Диалекты замещающего текста: .NET, Java

Первое регулярное выражение поддерживается только диалектом .NET из-за квантификатора ‹+› в ретроспективной проверке. Второе регулярное выражение можно использовать в диалектах .NET и Java, потому что диалект Java поддерживает конечные квантификаторы внутри ретроспективных проверок – даже достаточно длинные интервальные квантификаторы, такие как {1,100}. То есть версия только для .NET будет правильно работать с любыми числами, а версия для Java будет препятствовать добавлению запятых только при работе с числами, содержащими не более 100 цифр после десятичной точки. Второе число в квантификаторе ‹{1,100}› можно увеличить, если возникнет потребность обрабатывать числа с большим количеством цифр после десятичной точки.

В обоих регулярных выражениях новая ретроспективная проверка помещена в конец шаблона. У кого-то интуитивно может возникнуть желание реструктурировать регулярное выражение и перенести ретроспективную проверку в начало, но мы отказались от такого решения из соображений эффективности. Так как ретроспективная проверка является самой медленной частью регулярного выражения, размещение ее в конце позволяет регулярному выражению быстрее терпеть неудачу

в испытуемых строках, где до ретроспективной проверки дело может просто не дойти.

Поиск с заменой внутри совпавших чисел. При использовании других диалектов, отличных от .NET и Java, отсутствует возможность заглядывать назад в испытуемой строке на произвольную глубину, но тем не менее вы все же можете использовать ретроспективную проверку фиксированной длины для выявления чисел, которым не предшествует десятичная точка. Это позволит идентифицировать числа, в которые можно добавлять запятые (и корректно исключать любые последовательности цифр, следующие за десятичной точкой), но, так как ей будет соответствовать число целиком, нельзя просто включить запятую в замещающий текст.

В окончательном решении требуется использовать два регулярных выражения. Внешнее выражение будет отыскивать числа, в которые следует добавить запятые, а внутреннее – выполнять поиск позиций для вставки запятых внутри найденных чисел, в ходе операции поиска с заменой.

Внешнее регулярное выражение:

```
\b(?<!\.)[0-9]{4,}
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Это регулярное выражение совпадает с числами целиком, содержащими четыре и более цифр, которым не предшествует точка. Проверка границы слова в начале регулярного выражения гарантирует, что совпавшее число будет находиться в начале строки или будет отделяться от других чисел и слов. В противном случае регулярное выражение совпадало бы, например, с фрагментом 2345 в числе 0.12345. Иными словами, без проверки границы слова совпадение может начинаться со второй цифры после десятичной точки, так как точка не предшествует цифре в этой позиции.

Внутреннее регулярное выражение и замещающий текст остаются такими же, как в разделе «Простое решение» выше.

Замена совпадений, найденных внутренним регулярным выражением в каждом совпадении с внешним регулярным выражением, должна выполняться с помощью программного кода, а не с применением простой строки замещающего текста. При таком подходе мы сможем использовать внутреннее регулярное выражение внутри кода, выполняющего поиск чисел, пригодных для замены, с помощью внешнего регулярного выражения. На первый взгляд предлагаемое решение кажется сложным, однако все языки программирования, охватываемые этой книгой, делают его реализацию достаточно прямолинейной.

Ниже приводится законченное решение для Ruby 1.9:

```
subject.gsub(/\b(?<!\.)[0-9]{4,}/) {|match|
    match.gsub(/[0-9](?=(?:[0-9]{3})+(?![0-9]))/, '\0,')
}
```

Переменная subject в этом фрагменте хранит испытуемый текст, где требуется выполнить подстановку запятых. Строковый метод gsub в Ruby выполняет глобальный поиск с заменой. Помощь в реализации этого решения на других языках программирования можно получить в рецепте 3.16, где описывается, как выполнить замену совпадений фрагментами, сгенерированными в программном коде. Там приводятся примеры, демонстрирующие реализацию описываемого приема на каждом языке.

Отсутствие поддержки ретроспективных проверок в JavaScript и Ruby 1.8 ухудшает переносимость этого решения из-за использования ретроспективной проверки во внешнем регулярном выражении. Эту проблему можно решить в JavaScript и Ruby 1.8 включением в совпадение символа, предшествующего числу, если таковой имеется, и потребовать, чтобы он отличался от цифры и точки. Этот символ можно затем вставить обратно в текст, используя обратную ссылку в сгенерированном замещающем тексте.

Ниже приводится решение на JavaScript, реализующее этот прием:

```
subject.replace(/(^|[^0-9.])([0-9]{4,})/g, function($0, $1, $2) {
    return $1 + $2.replace(/[0-9](?=(?:[0-9]{3})+(?![0-9]))/g, "$&,");
});
```

См. также

Рецепт 6.11, где объясняется, как находить числа, в которых уже присутствует запятая, используемая для разделения групп разрядов.

Все остальные рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

6.13. Римские числа

Задача

Необходимо обеспечить совпадение с римскими числами, такими как IV, XIII и MVIII.

Решение

Римские числа без проверки на корректность:

```
^[MDCLXVI]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Современные римские числа, строгое соответствие:

```
^(?=[MDCLXVI])M*(C[MD]|D?C{0,3})(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Современные римские числа, гибкое соответствие:

```
^(?=[MDCLXVI])M*(C[MD]|D?C*)(X[CL]|L?X*)(I[XV]|V?I*)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Простые римские числа:

```
^(?=[MDCLXVI])M*D?C{0,4}L?X{0,4}V?I{0,4}$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Римские числа записываются с помощью символов M, D, C, L, X, V и I, представляющих значения 1000, 500, 100, 50, 10, 5 и 1 соответственно. Первому регулярному выражению соответствуют любые строки, состоящие из этих символов, без проверки порядка следования и количества символов, допустимых в римских числах.

В наше время (имеются в виду последние несколько сотен лет) запись римских чисел регулируется сводом строгих правил. Эти правила обеспечивают точное положение римских цифр в числах. Например, 4 всегда записывается как IV и никогда как IIII.

Второе регулярное выражение в решении соответствует только римским числам, которые следуют этим правилам.

Каждая ненулевая цифра в десятичном представлении в римской форме записи записывается отдельно. Число 1999 записывается как MCMXCIX, где M – это 1000, CM – это 900, XC – это 90 и IX – это 9. Это число нельзя записать как MIM или IMM.

Тысячи записываются просто: по одному символу M на каждую тысячу, что легко можно выразить как `‹M*›`.

Сотни могут записываться 10 разными способами, совпадения с которыми мы обеспечили с помощью конструкций выбора. Выражение `‹C[MD]›` соответствует формам записи CM и CD, которые представляют числа 900 и 400. `‹D?C{0,3}›` соответствует формам записи DCCC, DCC, DC, D, CCC, CC, C и пустой строке, представляющим 800, 700, 600, 500, 300, 200, 100 и «ничто». Это дает нам все 10 цифр в разряде сотен.

Совпадение с десятками обеспечивает выражение `‹X[CL]|L?X{0,3}›`, а с единицами – выражение `‹I[XV]|V?I{0,3}›`. В них используется тот же синтаксис, но используются другие символы.

Все четыре части регулярного выражения могут быть необязательными, потому что в любом из разрядов может быть ноль. У римлян не было символа или слова, представляющего ноль. Поэтому ноль в римских числах не записывается. Несмотря на то что каждая часть римского числа сама по себе может быть необязательной, тем не менее все они не могут быть необязательными одновременно. Мы должны гарантировать невозможность совпадений нулевой длины с регулярным выражением. Для этого мы добавили в начало регулярного выражения опережающую проверку ‹(?=[MDCLXVI])›. Эта опережающая проверка, как описывается в рецепте 2.16, проверяет возможность совпадения хотя бы с одним символом. Опережающая проверка не поглощает символы, поэтому совпадения с ней могут повторно совпасть с остальной частью регулярного выражения.

Третье регулярное выражение немного более гибкое. Оно допускает наличие таких цифр, как IIII, и при этом принимает IV.

Четвертое регулярное выражение соответствует римским числам, которые записаны без применения операции вычитания, из-за чего все символы должны быть записаны в порядке убывания их значений. Число 4 должно записываться как IIII, а не как IV. Сами римляне записывали числа именно таким способом.

Все регулярные выражения заключены в якорные метасимволы (рецепт 2.5), чтобы обеспечить проверку, является ли весь испытуемый текст римским числом, в противовес поиску римского числа в текстовом документе. Если потребуется выполнить поиск римского числа в объемном тексте, можно заменить ‹^› и ‹$› на ‹\b›.

Преобразование римских чисел в десятичные

Следующая функция на языке Perl использует «строгое» регулярное выражение из этого рецепта, чтобы убедиться, что полученная ею строка является римским числом. Затем она использует регулярное выражение ‹[MDLV]|C[MD]?|X[CL]?|I[XV]?›, чтобы обойти в цикле все цифры в числе и сложить их значения:

```
sub roman2decimal {
    my $roman = shift;
if ($roman =~
    m/^(?=[MDCLXVI])
      (M*)                # 1000
      (C[MD]|D?C{0,3}) # 100
      (X[CL]|L?X{0,3}) # 10
      (I[XV]|V?I{0,3}) # 1
      $/ix)
{
    # Найдено римское число
    my %r2d = ('I' =>    1, 'IV' =>   4, 'V' =>   5, 'IX' =>   9,
```

```
                    'X' =>   10, 'XL' =>  40, 'L' =>  50, 'XC' =>  90,
                    'C' =>  100, 'CD' => 400, 'D' => 500, 'CM' => 900,
                    'M' => 1000);
        my $decimal = 0;
        while ($roman =~ m/[MDLV]|C[MD]?|X[CL]?|I[XV]?/ig) {
            $decimal += $r2d{uc($&)};
        }
        return $decimal;
    } else {
        # Это не римское число
        return 0;
    }
}
```

См. также

Все остальные рецепты в этой главе демонстрируют разные способы поиска различных типов чисел с применением регулярных выражений.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

В исходном программном коде, который приводится в этом рецепте, используется прием обхода всех совпадений, обсуждавшийся в рецепте 3.11.

7

Исходный код и файлы журналов

Как было показано в рецепте 3.22, регулярные выражения отлично подходят для парсинга исходного текста и с успехом могут использоваться для конструирования парсеров, выполняющих разбор файлов в нестандартных форматах или с исходным кодом на языках сценариев. В этой главе приводится множество рецептов выделения синтаксических элементов, широко используемых в языках программирования и в других текстовых форматах. Регулярные выражения из этих рецептов можно объединять в более крупные регулярные выражения для использования в парсере. Эти регулярные выражения могут также пригодиться при редактировании исходного кода в текстовом редакторе и для поиска с помощью инструмента grep.

Во второй части этой главы показано, как с помощью регулярных выражений извлекать информацию из файлов журналов. В основном в рецептах будут использоваться файлы журналов веб-сервера, так как они доступны большинству читателей и многие даже могут быть знакомы с их форматом. Приемы, демонстрируемые в этих рецептах, легко можно распространить на любые другие форматы файлов журналов.

7.1. Ключевые слова

Задача

Многим читателям приходится работать с файлами форм, используемыми в прикладных программах. Слова «end», «in», «inline», «inherited», «item» и «object» являются зарезервированными в этом формате.[1]

---

[1] Идеей для этого рецепта послужили файлы форм в Delphi, в которых используются именно эти ключевые слова, за исключением «in», которое было добавлено здесь для иллюстрации некоторых ловушек.

Необходимо написать регулярное выражение, совпадающее с любым из этих ключевых слов.

Решение

Простое решение выглядит достаточно прямолинейным и может использоваться со всеми диалектами, описываемыми в этой книге:

```
\b(?:end|in|inline|inherited|item|object)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Для диалектов, поддерживающих атомарную группировку, это регулярное выражение можно оптимизировать, как показано ниже:

```
\b(?>end|in(?:line|herited)?|item|object)\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Реализовать в регулярном выражении сопоставление со словом из списка очень просто. Достаточно использовать конструкцию выбора. Присутствие метасимволов границы слова в начале и в конце регулярного выражения гарантируют совпадение с целым словом. Регулярное выражение должно находить совпадение `inline`, а не `in`, если файл содержит ключевое слово `inline`; оно также должно терпеть неудачу при сопоставлении со словом `interesting`. Так как оператор выбора имеет самый низкий приоритет среди всех операторов регулярных выражений, мы поместили список ключевых слов в группу. Здесь для большей эффективности используется несохраняющая группа. При использовании этого регулярного выражения в составе более крупного регулярного выражения, напротив, может оказаться желательным использовать сохраняющую группировку, чтобы иметь возможность определить, действительно ли регулярное выражение совпало с ключевым словом или с чем-то иным.

В диалектах, поддерживающих атомарную группировку, это регулярное выражение можно оптимизировать. Когда первое выражение из раздела «Решение» встречает слово `interesting`, оно обнаруживает совпадение с альтернативой `‹in›`. Затем граница слова в конце выражения вынуждает его потерпеть неудачу. Далее механизм регулярных выражений выполнит возврат, тщетно пытаясь опробовать остальные альтернативы.

Заключив конструкцию выбора внутрь атомарной группировки, мы препятствуем механизму регулярных выражений выполнить возврат после неудачи сопоставления со второй границей слова `‹\b›`. Благодаря этому скорость сопоставления увеличивается.

Так как регулярное выражение не будет выполнять возвраты, его необходимо скомпоновать так, чтобы они и не потребовались для обнаруже-

ния совпадения с любым ключевым словом. Когда первое регулярное выражение встречает слово inline, оно сначала обнаруживает совпадение с in. Затем попытка сопоставления со второй границей слова терпит неудачу. Механизм регулярных выражений выполняет возврат, находит совпадение с альтернативой inline и второй границей слова, в результате чего обнаруживается совпадение со всем регулярным выражением. При использовании атомарной группировки возвраты выполняться не будут, поэтому фрагмент конструкции выбора ‹in|inline|inherited› в первом регулярном выражении пришлось преобразовать в ‹in(?:line|herited)?› во втором регулярном выражении. Первое регулярное выражение пытается найти совпадения in, inline и inherited именно в таком порядке, потому что оператор выбора отличается «нетерпеливостью». Второе регулярное выражение пытается сначала найти совпадение с inline или inherited благодаря максимальному квантификатору и только потом проверяет совпадение с in. Только после обнаружения совпадения с inline, inherited или in второе регулярное выражение выполняет сопоставление со второй границей слова. Если граница слова не обнаружена, нет смысла возвращаться, чтобы опробовать другие альтернативы, что мы и выразили с помощью атомарной группировки.

Варианты

Простого сопоставления с ключевыми словами может оказаться недостаточно. Эти слова не интерпретируются как зарезервированные в файлах форм, если они заключены в апострофы. Если форма содержит элемент управления с заголовком «The end is near», в файле формы он будет храниться в следующем виде:

```
object Button1: TButton
    Caption = 'The end is near'
end
```

В этом фрагменте второе вхождение end является ключевым словом, а первое – нет. Чтобы только второе вхождение end интерпретировалось как ключевое слово, необходимо более сложное решение.

Нет простого способа заставить регулярное выражение совпадать только с ключевыми словами, находящимися не внутри строки. Зато легко можно заставить его совпадать и со строками, и с ключевыми словами.

```
\b(end|in|inline|inherited|item|object)\b|'[^'\r\n]*(?:''[^'\r\n]*)*'
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Когда регулярное выражение встречает апостроф, оно совпадает со всей строкой, вплоть до следующего апострофа. Следующая попытка сопоставления будет выполняться после конца строки. Благодаря этому регулярное выражение не будет искать совпадения с ключевыми словами внутри строк, вместо этого оно будет совпадать с целыми строками.

В предыдущем примере регулярное выражение сначала найдет совпадение `object`, затем `'The end is near'` и наконец `end` в конце испытуемого текста.

Чтобы иметь возможность определить, с чем именно совпало регулярное выражение, со строкой или с ключевым словом, для списка ключевых слов потребовалось использовать сохраняющую группу вместо несохраняющей. Когда регулярное выражение совпадает с ключевым словом, оно будет сохранять совпадение с первой (и единственной) сохраняющей группой. При совпадении со строкой первая сохраняющая группа останется пустой, так как она не будет участвовать в сопоставлении в этом случае.

При конструировании парсеров, как описывалось в рецепте 3.22, практически всегда приходится объединять регулярное выражение для поиска ключевых слов с регулярными выражениями для поиска строк и всех остальных лексем, встречающихся в файлах данного формата. При этом используется тот же прием, что применялся здесь для ключевых слов и строк. Просто регулярное выражение будет иметь намного больше альтернатив, охватывающих весь синтаксис формата файлов. Описанный прием помогает автоматически избежать совпадений с ключевыми словами внутри строк.

При поиске совпадений с ключевыми словами в других форматах файлов или языках программирования сопоставления с границами слов может оказаться недостаточно. Например, во многих языках `$end` является именем переменной, а `end` – ключевым словом. В таких ситуациях проверки границ слов недостаточно, чтобы гарантировать отсутствие совпадения с ключевым словом там, где его не должно быть. Выражение `‹\bend\b›` совпадет с `end` в идентификаторе `$end`. Знак доллара не является символом слова, а буквы являются. Метасимвол `‹\b›` найдет совпадение между знаком доллара и буквой.

Решить эту проблему можно с помощью ретроспективной проверки. Чтобы убедиться в отсутствии знака доллара перед ключевым словом, в выражении `‹(?<![$\w])(?:end|in|inline|inherited|item|object)\b›`, например, используется негативная ретроспективная проверка. Негативная ретроспективная проверка включает `‹\w›`, и нам все еще необходимо использовать проверку совпадения с границей слова `‹\b›` в конце, чтобы убедиться, что найденное ключевое слово не является частью более длинного слова.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.14 объясняются особенности атомарной группировки, а в рецепте 2.12 описываются приемы использования квантификаторов для оптимизации регулярных выражений. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

7.2. Идентификаторы

Задача

Требуется написать регулярное выражение, совпадающее с любым идентификатором в исходном программном коде. Синтаксис языка программирования требует, чтобы идентификаторы начинались с символа подчеркивания или буквы ASCII. Последующими символами в идентификаторе могут быть символы подчеркивания, буквы ASCII или цифры. Идентификатор может иметь длину от 1 до 32 символов.

Решение

```
\b[a-z_][0-9a-z_]{0,31}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Символьный класс ‹[a-z_]› совпадает с первым символом в идентификаторе, а символьный класс ‹[0-9a-z_]› – со вторым и последующими символами. Регулярное выражение допускает наличие от 0 до 31 последующих символов. Здесь вместо ‹\w› используется ‹[0-9a-z_]›, что избавляет от необходимости беспокоиться по поводу совпадения сокращения ‹\w› с символами, не входящими в набор ASCII. Мы не включили буквы верхнего регистра в символьный класс, потому что тот же эффект дает включение параметра нечувствительности к регистру символов, а сам символьный класс оказывается немного короче. Чтобы исключить зависимость от параметра нечувствительности к регистру, можно использовать выражение ‹\b[a-zA-Z_][0-9a-zA-Z_]{0,31}\b›.

Наличие двух границ слова ‹\b› гарантирует, что регулярное выражение не совпадет с последовательностью из более чем 32 алфавитно-цифровых символов.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов.

7.3. Числовые константы

Задача

Требуется написать регулярное выражение, совпадающее с десятичными целыми числами, не начинающимися с нуля; с восьмеричными целыми числами, начинающимися с нуля; с шестнадцатеричными числами, начинающимися с префикса 0x; с двоичными числами, начинаю-

щимися с префикса `0b`. Целые числа могут заканчиваться символом `L`, обозначающим длинное целое значение.

Регулярное выражение должно иметь отдельные (именованные) сохраняющие группы для десятичных, восьмеричных, шестнадцатеричных и двоичных чисел, сохраняемых без каких-либо префиксов или окончаний, чтобы программный код, использующий это регулярное выражение, легко мог определить систему счисления и преобразовать текст в фактическое число. Для чисел с окончанием `L` также должна быть отдельная сохраняющая группа, чтобы упростить идентификацию типа целого числа.

Решение

```
\b(?:(?<dec>[1-9][0-9]*)
| (?<oct>0[0-7]*)
| 0x(?<hex>[0-9A-F]+)
| 0b(?<bin>[01]+)
)(?<L>L)?\b
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?:(?P<dec>[1-9][0-9]*)
| (?P<oct>0[0-7]*)
| 0x(?P<hex>[0-9A-F]+)
| 0b(?P<bin>[01]+)
)(?P<L>L)?\b
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4, Perl 5.10, Python

```
\b(?:([1-9][0-9]*)|(0[0-7]*)|0x([0-9A-F]+)|0b([01]+))(L)?\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Данное регулярное выражение фактически является комбинацией решений, представленных в рецептах 6.5 (десятичные числа), 6.4 (восьмеричные числа), 6.2 (шестнадцатеричные числа) и 6.3 (двоичные числа). Единственная цифра «ноль» может быть и десятичным, и восьмеричным числом. Однако это не имеет никакого значения, потому что во всех системах счисления цифра ноль всегда представляет число ноль. Поэтому из регулярного выражения, совпадающего с десятичными числами, была удалена та его часть, которая соответствует числу ноль.

Все четыре альтернативы заключены в несохраняющую группу, чтобы гарантировать, что границы слова и окончание `L` будут применяться к выражению в целом, а не только к первой и последней альтернативам.

Именованные сохраняющие группы упрощают чтение регулярного выражения и преобразование совпавшего фрагмента в фактическое число в программном коде. Диалекты JavaScript и Ruby 1.8 не поддерживают именованное сохранение. Для этих языков предлагается альтернативное решение на основе пяти нумерованных сохраняющих групп.

См. также

В главе 6 во всех подробностях обсуждается тема сопоставления с целыми и вещественными числами. Помимо приемов, описанных там, в этом рецепте использовалось именованное сохранение (рецепт 2.11) и режим свободного форматирования (рецепт 2.18).

7.4. Операторы

Задача

Для реализации схемы подсветки синтаксиса в текстовом редакторе потребовалось написать регулярное выражение, совпадающее с любыми символами, которые могут использоваться в качестве операторов в языке программирования, для которого создается схема: -, +, *, /, =, <, >, %, &, ^, |, !, ~ и ?. Регулярное выражение не должно проверять, являются ли комбинации символов допустимыми операторами. Это не является задачей схемы подсветки синтаксиса – она просто должна подсвечивать все символы операторов.

Решение

```
[-+*/=<>%&^|!~?]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Для тех, кто прочитал рецепт 2.3, решение будет очевидным. Кого-то может удивить, почему это задание было включено в виде отдельного рецепта.

В центре внимания этой главы находятся регулярные выражения, которые можно использовать в крупных системах, таких как схемы подсветки синтаксиса. Подобные системы часто объединяют регулярные выражения с использованием оператора выбора, что может приводить к появлению ловушек, не всегда очевидных, если рассматривать регулярное выражение в изоляции.

Одна из таких ловушек состоит в том, что система, использующая данное регулярное выражение, наверняка будет использовать другие регулярные выражения, совпадающие с теми же символами. Во многих языках

программирования символ / используется как оператор деления, а пара символов // начинает комментарий. Если объединить регулярное выражение из этого рецепта с регулярным выражением из рецепта 7.5 в выражение ‹(?<operator>[-+*/=<>%&^|!~?])|(?<comment>//.*)›, можно будет обнаружить, что система не находит комментарии. Все символы слэша будут распознаваться как операторы.

Проблему можно исправить, если поменять альтернативы местами: ‹(?<comment>//.*)|(?<operator>[-+*/=<>%&^|!~?])›. Это регулярное выражение всегда будет совпадать с двумя идущими подряд слэшами как с однострочным комментарием. Попытка найти совпадение с каким-либо оператором не будет выполняться, пока первая половина регулярного выражения не потерпит неудачу. Если у вас имеется приложение, объединяющее несколько регулярных выражений, такое как текстовый редактор с подсветкой синтаксиса на основе регулярных выражений, вам потребуется узнать, в каком порядке это приложение объединяет регулярные выражения.

Еще одна ловушка кроется в попытке «оптимизировать» регулярное выражение путем добавления квантификатора после символьного класса: ‹[-+*/=<>%&^|!~?]+›. Так как от системы подсветки синтаксиса требуется обеспечить подсветку всех символов операторов, более эффективным выглядит решение подсветить сразу все операторы, следующие друг за другом. И такое решение было бы оправданным, если бы подсветка операторов была единственной обязанностью схемы. Но оно не дает желаемого результата в некоторых ситуациях, даже когда регулярные выражения объединяются в правильном порядке, как было определено в предыдущем абзаце: ‹(?<comment>//.*)|(?<operator>[-+*/=<>%&^|!~?]+)›. Это регулярное выражение будет правильно подсвечивать операторы и однострочные комментарии, кроме случая, когда однострочный комментарий следует сразу за оператором. Когда регулярное выражение встретит последовательность символов !//bang, альтернатива «comment» потерпит неудачу при попытке найти совпадение с *. Затем регулярное выражение попытается сопоставить альтернативу «operator». Но она совпадет не только с !; в совпадение попадут три символа !//, потому что квантификатор ‹+› после символьного класса постарается найти максимально большое количество символов операторов. После обнаружения этого совпадения регулярное выражение вновь попытается найти совпадение в оставшейся части строки bang и потерпит неудачу, потому что символы, начинающие однострочный комментарий, уже были поглощены предыдущим совпадением.

Если отбросить квантификатор и использовать выражение ‹(?<comment>//.*)|(?<operator>[-+*/=<>%&^|!~?])›, альтернатива «operator» совпадет только с символом ! в строке !//bang, а в следующей попытке сопоставления альтернатива «comment» совпадет с остатком строки //bang.

7.5. Однострочные комментарии

Задача

Необходимо написать регулярное выражение, совпадающее с комментарием, начинающимся с пары символов // и продолжающимся до конца строки.

Решение

```
//.*
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Символ прямого слэша не имеет специального значения в регулярных выражениях, поэтому совпадение с началом комментария можно обеспечить двумя литеральными символами «//». В некоторых языках программирования символы слэша применяются для ограничения регулярных выражений. При использовании этого регулярного выражения в программном коде может понадобиться экранировать слэши, как описывается в рецепте 3.1.

Подвыражение «.*» совпадает с любой последовательностью символов до конца строки. Здесь не требуется предпринимать какие-либо действия, чтобы заставить регулярное выражение останавливаться в конце строки. Достаточно убедиться, что параметр «точке соответствуют границы строк» был выключен перед применением этого регулярного выражения.

См. также

Рецепт 2.4, где говорится, что точке соответствуют любые символы.

7.6. Многострочные комментарии

Задача

Необходимо написать регулярное выражение, совпадающее с комментарием, начинающимся с пары символов /* и заканчивающимся парой символов */. Вложенные комментарии не допускаются. Любые пары символов /* между /* и */ должны интерпретироваться как часть комментария. Комментарии могут занимать несколько строк в тексте.

Решение

```
/\*.*?\*/
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
/\*[\s\S]*?\*/
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Символ прямого слэша не имеет специального значения в регулярных выражениях, а символ звездочки имеет. Поэтому необходимо экранировать звездочку обратным слэшем. В результате получаются подвыражения ‹/*› и ‹*/›, соответствующие парам символов /* и */. Обратные и/или прямые слэши могут иметь специальное значение в строковых литералах, в программном коде, поэтому может понадобиться экранировать слэши, как описывается в рецепте 3.1.

Для совпадения со всем, что находится между двумя ограничителями комментария, используется подвыражение ‹.*?›. Параметр «точке соответствуют границы строк», поддерживаемый большинством диалектов регулярных выражений, обеспечит распространение этого подвыражения на несколько строк. Чтобы регулярное выражение прекратило сопоставление при встрече первой, а не последней в файле пары символов */, следующей за /*, здесь используется минимальный квантификатор.

JavaScript – единственный диалект из обсуждаемых в этой книге, не поддерживающий параметр «точке соответствуют границы строк». Если описываемую задачу потребуется решить на JavaScript без использования библиотеки XRegExp, для достижения того же эффекта можно использовать символьный класс ‹[\s\S]›. Несмотря на то что этот символьный класс можно использовать и в других диалектах, поступать так не рекомендуется, так как механизмы регулярных выражений обычно оптимизируют сопоставление с точкой, которая является одной из простейших особенностей регулярных выражений.

Варианты

Если регулярное выражение предполагается использовать в системе, так или иначе связанной с редактированием исходного программного кода, может потребоваться сделать необязательной пару символов, закрывающих комментарий. В этом случае все, от начала комментария и до конца файла, будет интерпретироваться как комментарий, пока не будет введена пара символов */. Например, механизм подсветки синтаксиса в текстовых редакторах обычно действует именно так. Если сделать необязательной пару символов, закрывающих комментарий, это никак не повлияет на интерпретацию файлов, где все многострочные комментарии корректно закрыты. Квантификатор для закрывающей пары символов должен быть максимальным, чтобы он обнаруживал совпадение, если таковое имеется. Квантификатор для точки должен быть минимальным, чтобы сопоставление прекращалось при встрече первой же пары закрывающих символов.

```
/\*.*?(?:\*/)?
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
/\*[\s\S]*?(?:\*/)?
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Рецепт 2.4, гда рассказывается о точке, а также о параметре, обеспечивающем ее совпадение с границами строк, и обходных решениях для JavaScript. В рецепте 2.13 описываются различия между максимальными и минимальными квантификаторами.

7.7. Все комментарии

Задача

Необходимо написать регулярное выражение, совпадающее с комментариями обоих типов, однострочными и многострочными, как описывается в разделах «Задача» в двух предыдущих рецептах.

Решение

```
(?-s://.*)|(?s:/\*.*?\*/)
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl

```
(?-m://.*)|(?m:/\*.*?\*/)
```

Параметры: нет
Диалект: Ruby

```
//[^\r\n]*|/\*.*?\*/
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
//.*|/\*[\s\S]*?\*/
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

На первый взгляд кажется, что достаточно объединить решения из двух предыдущих рецептов с помощью оператора выбора: `<//.*|/\*.*?\*/>`. Однако этот прием не даст желаемого результата, потому что для первой альтернативы параметр «точке соответствуют границы строк» должен быть сброшен, а для второй, напротив, установлен. В ситуациях, когда необходимо объединить два регулярных выражения, использую-

щих точку, следует использовать модификатор режима для включения параметра «точке соответствуют границы строк» во второй половине регулярного выражения. Решение, представленное здесь, дополнительно явно выключает этот параметр в первой половине выражения. Строго говоря, в этом нет необходимости, но такой подход более ясно отражает намерения и помогает предотвратить ошибки с параметром «точке соответствуют границы строк», если данное регулярное выражение потребуется внедрить в еще более длинное регулярное выражение.

Диалекты Python и JavaScript (ни с библиотекой XRegExp, ни без нее) не поддерживают модификаторы режимов в середине регулярных выражений. При использовании диалектов Python и JavaScript с применением XRegExp в части выражения сопоставления с однострочным комментарием совпадение со всем до конца строки можно обеспечить с помощью инвертированного символьного класса `‹[^\r\n]*›`, а в части сопоставления с многострочным комментарием использовать подвыражение `‹.*?›` с включенным параметром «точке соответствуют границы строк».

При использовании диалекта JavaScript без применения XRegExp нет возможности обеспечить совпадение точки с границами строк. Поэтому в этой версии решения в части выражения сопоставления с однострочным комментарием было оставлено подвыражение `‹.*›`, а в части сопоставления с многострочным комментарием используется подвыражение `‹[\s\S]*?›`.

См. также

Рецепт 2.4, в котором рассказывается о точке, а также о параметре, обеспечивающем ее совпадение с границами строк, и обходных решениях для JavaScript. В рецепте 2.8 описывается применение оператора выбора.

7.8. Строки

Задача

Необходимо написать регулярное выражение, совпадающее со строками – последовательностями из нуля и более символов, заключенными в кавычки. Строка, в которой между кавычками ничего нет, – это пустая строка. Две кавычки в строке символов, следующие друг за другом, обозначают один символ – символ кавычки. Строки не могут содержать символы конца строки. Ни обратные слэши, ни какие-то другие символы внутри строк не могут иметь специальное значение.

Регулярное выражение должно совпадать с любыми строками, включая пустые, и возвращать единое совпадение при встрече в строке двух кавычек, следующих подряд. Например, оно должно возвращать `"before quote""after quote"` как единое совпадение, а не два: `"before quote"` и `"after quote"`.

Решение

```
¨[^¨\r\n]*(?:""[^¨\r\n]*)*¨
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Совпадение со строкой, не содержащей кавычек или символов конца строки, легко обеспечить с помощью выражения ‹¨[^\r\n¨]*¨›. Кавычки в регулярных выражениях интерпретируются буквально, поэтому шаблон строки как последовательности символов, не содержащей кавычек или символов конца строки, можно выразить с помощью инвертированного символьного класса.

Но по условиям задачи строка может содержать кавычки, если они следуют парами. Сопоставление с такими парами кавычек выразить совсем несложно, если обрабатывать их отдельно. Вслед за открывающей кавычкой следует символьный класс ‹[^\r\n¨]*›, совпадающий с любыми символами, кроме кавычек или символов конца строки. За ними может следовать ноль или более пар кавычек. Совпадение с парами кавычек можно было бы описать с помощью выражения ‹(?:¨¨)*›, но после каждой пары кавычек в строке могут следовать другие символы, не являющиеся ни кавычками, ни символами конца строки. Поэтому для одной пары кавычек и следующих за ней других символов, не кавычек и не символов конца строки, выполняется сопоставление с помощью выражения ‹¨¨[^\r\n¨]*›, а для всех пар — с помощью подвыражения ‹(?:¨¨[^\r\n¨]*)*›. Регулярное выражение заканчивается кавычкой, закрывающей строку.

Совпадение, возвращаемое регулярным выражением, будет содержать строку целиком, включая ограничивающие ее кавычки и пары кавычек внутри строки. Чтобы получить только содержимое строки, программный код, использующий регулярное выражение, должен будет выполнить дополнительные операции. Во-первых, он должен будет удалить кавычки в начале и в конце совпадения. Затем он должен отыскать все пары кавычек и заменить каждую из них одной кавычкой.

У кого-то может возникнуть вопрос: почему бы не использовать более простое регулярное выражение ‹¨(?:[^¨\r\n]|¨¨)*¨›? Это регулярное выражение соответствует паре кавычек, содержащих ноль или более вхождений совпадений с любой из двух альтернатив ‹(?:[^¨\r\n]|¨¨)*›. Символьный класс ‹[^¨\r\n]› соответствует символу, не являющемуся кавычкой или символом конца строки. Выражение ‹¨¨› соответствует паре кавычек. В результате их объединения получается регулярное выражение, соответствующее паре кавычек, содержащих ноль или более символов, не являющихся кавычками, или символами конца строки, или парами кавычек. Это определение соответствует описанию строки в формулировке задачи. Действительно, это регулярное выражение коррект-

но будет совпадать с нужными нам строками, но оно неэффективно. Механизм регулярных выражений будет входить в группу с двумя альтернативами при сопоставлении с каждым символом в строке. При сопоставлении с выражением из раздела «Решение» механизм регулярных выражений будет входить в группу только для каждой пары кавычек, встречающихся в строках достаточно редко.

Можно попробовать оптимизировать неэффективное выражение так: ‹"(?:[^"\r\n]+|"")*"›. Суть в том, чтобы позволить этому регулярному выражению входить в группу только для каждой пары кавычек и для каждой последовательности символов без кавычек или символов конца строки. Да, эта оптимизация будет давать положительный эффект при сопоставлении регулярного выражения с допустимыми строками. Но если где-нибудь в файле встретится строка без закрывающей кавычки, тут же возникнет проблема катастрофических возвратов. Если попытка найти совпадение с закрывающей кавычкой потерпит неудачу, механизм регулярных выражений будет пытаться опробовать все перестановки, допускаемые квантификаторами ‹+› и ‹*›, сопоставляя все символы между открывающей кавычкой и концом строки.

В табл. 7.1 показано, как это регулярное выражение будет опробовать все возможные способы сопоставления для строки "abcd. В ячейках таблицы приводится текст, совпавший с подвыражением ‹[^"\r\n]+›. Сначала оно обнаруживает совпадение abcd, но когда попытка сопоставления с закрывающей кавычкой потерпит неудачу, квантификатор ‹+› выполнит возврат, уступив часть своего совпадения. Когда это произойдет, квантификатор ‹*› повторит сопоставление с группой, инициировав следующую итерацию сопоставления подвыражения ‹[^"\r\n]+› с оставшимися символами. Теперь у нас имеется две итерации, которые будут выполнять возвраты. Так будет продолжаться до тех пор, пока в каждой итерации сопоставления с ‹[^"\r\n]+› не останется по одному символу, и квантификатор ‹*› повторит сопоставление с группой столько раз, сколько символов в строке.

Таблица 7.1. Разделители строк

Перестановка	1-я итерация ‹[^"\r\n]+›	2-я итерация ‹[^"\r\n]+›	3-я итерация ‹[^"\r\n]+›	4-я итерация ‹[^"\r\n]+›
1	abcd	неприменимо	неприменимо	неприменимо
2	abc	d	неприменимо	неприменимо
3	ab	cd	неприменимо	неприменимо
4	ab	c	d	неприменимо
5	a	bcd	неприменимо	неприменимо
6	a	bc	d	неприменимо
7	a	b	cd	неприменимо
8	a	b	c	d

Как видите, число перестановок растет экспоненциально[1] относительно числа символов после открывающей кавычки. При обработке файлов с короткими строками это приведет к замедлению работы приложения. При обработке файлов с очень длинными строками приложение может «зависнуть» или завершиться аварийно. Если для сопоставления с множеством строк использовать вариант `‹"(?:[^"]+|"")*">`, перестановками может оказаться охвачена вся оставшаяся часть файла, если дальше нигде не встретится кавычка.

Можно было бы попытаться предотвратить возвраты, применив атомарную группировку `‹"(?>[^"\r\n]+|"")*">` или захватывающие квантификаторы `‹"(?:[^"\r\n]++|"")*+">`, если используемый диалект регулярных выражений поддерживает их. Но необходимость использовать специальные возможности сводит на нет все попытки создать более простое регулярное выражение, чем представленное в разделе «Решение».

Варианты

Ничуть не сложнее выглядит реализация сопоставления со строками в апострофах:

```
'[^'\r\n]*(?:''[^'\r\n]*)*'
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если язык программирования поддерживает строки и в апострофах, и в кавычках, сопоставление с ними необходимо оформить в виде отдельных альтернатив:

```
"[^"\r\n]*(?:""[^"\r\n]*)*"|'[^'\r\n]*(?:''[^'\r\n]*)*'
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если строки могут включать символы конца строки, достаточно просто убрать их из инвертированного символьного класса:

```
"[^"]*(?:""[^"]*)*"
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если регулярное выражение предполагается использовать в системе, так или иначе связанной с редактированием исходного программного кода, может потребоваться сделать необязательной закрывающую кавычку. В этом случае все, от начала и до конца строки в тексте, будет интерпретироваться как строковый литерал, пока не будет введена закрывающая кавычка. Например, механизм подсветки синтаксиса в тексто-

---

[1] При наличии n символов между открывающей кавычкой и концом строки механизм регулярных выражений выполнит 2^{n-1} перестановок для сопоставления с `‹(?:[^"\r\n]+|"")*›`.

вых редакторах обычно действует именно так. Если сделать необязательной закрывающую кавычку, это никак не повлияет на работу регулярного выражения в файлах, где все строковые литералы корректно заключены в кавычки. Квантификатор для закрывающей кавычки должен быть максимальным, чтобы он обнаруживал совпадение, если таковое имеется. Инвертированные символьные классы устраняют риск ошибочного совпадения с закрывающей кавычкой как частью строкового литерала.

```
"[^"\r\n]*(?:""[^"\r\n]*)*"?
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

В рецептах 2.15 и 2.14 объясняется, что такое катастрофические возвраты и как их предупредить с помощью атомарной группировки и захватывающих квантификаторов.

7.9. Строки с экранированными символами

Задача

Необходимо написать регулярное выражение, совпадающее со строками – последовательностями из нуля и более символов, – заключенными в кавычки. Строка, в которой между кавычками ничего нет, – это пустая строка. В строку может быть включена кавычка, если она экранирована символом обратного слэша. Обратные слэши также могут использоваться для экранирования других символов в строке. Строки не могут содержать символы конца строки, и эти символы не могут экранироваться обратными слэшами.

Решение

```
"[^"\\\r\n]*(?:\\.[^"\\\r\n]*)"
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Это регулярное выражение имеет ту же структуру, что и выражение в предыдущем рецепте. Разница лишь в том, что теперь специальное

значение имеют два символа: кавычка и обратный слэш. В этом регулярном выражении совпадение с обоими символами исключается с помощью двух инвертированных символьных классов. Для сопоставления с каждым экранированным символом используется подвыражение ‹\\.›. Последовательность ‹\\› соответствует одному символу обратного слэша, а ‹.› соответствует любому символу, не являющемуся концом строки. Параметр «точке соответствуют границы строк» должен быть выключен.

Варианты

Совпадение со строками, заключенными в апострофы, определяется так же просто:

```
'[^'\\\r\n]*(?:\\.[^'\\\r\n]*)'
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если язык программирования поддерживает строки и в апострофах, и в кавычках, сопоставление с ними необходимо оформить в виде отдельных альтернатив:

```
"[^"\\\r\n]*(?:\\.[^"\\\r\n]*)"|'[^'\\\r\n]*(?:\\.[^'\\\r\n]*)'
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если строки могут включать символы конца строки, экранированные обратными слэшами, можно изменить первоначальное регулярное выражение, разрешив в нем совпадение с символами конца строки после обратного слэша. В следующем выражении используется конструкция ‹(?:.|\r?\n)› вместо параметра «точке соответствуют границы строк», чтобы гарантировать корректное совпадение с разрывами строк в стиле Windows. Точка может соответствовать только символу CR в последовательности CR LF, обозначающей конец строки, и регулярное выражение может терпеть неудачу при сопоставлении с символом LF. Подвыражение ‹\r?\n› корректно совпадает с разрывами строк и в стиле Windows, и в стиле UNIX.

```
"[^"\\\r\n]*(?:\\(?:.|\r?\n)[^"\\\r\n]*)"
```

Параметры: нет (параметр «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если строки могут включать даже неэкранированные символы конца строки, достаточно просто убрать их из инвертированных символьных классов. Параметр «точке соответствуют границы строк» должен быть включен.

```
"[^"\\]*(?:\\.[^"\\]*)*"
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Для стандартного диалекта JavaScript (без применения XRegExp) требуется отдельное решение, потому что в нем не поддерживается возможность совпадения точки с границами строк.

```
"[^"\\]*(?:\\[\s\S][^"\\]*)*"
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Рецепт 7.8, где описывается базовая структура регулярного выражения, предложенного в решении к этому рецепту.

В рецепте 2.4 рассказывается о точке, а также о параметре, обеспечивающем ее совпадение с границами строк, и обходных решениях для JavaScript.

7.10. Литералы регулярных выражений

Задача

Необходимо написать регулярное выражение, совпадающее с литералами регулярных выражений в исходном программном коде, чтобы можно было легко находить их в текстовом редакторе или с помощью инструмента grep. В соответствии с синтаксисом предполагаемого здесь языка программирования литералы регулярных выражений заключаются в символы прямого слэша. Прямые слэши в литералах регулярных выражений должны экранироваться обратными слэшами.

Требуемое регулярное выражение должно совпадать со всем, что выглядит как литерал регулярного выражения. В нем не требуется проверять текст между прямыми слэшами, чтобы убедиться, что он действительно представляет допустимое регулярное выражение.

Так как в этом рецепте требуется написать единственное регулярное выражение, а не полноценный компилятор, это регулярное выражение должно отличать ситуации, когда прямой слэш используется как оператор деления, а когда – как начальный символ литерала регулярного выражения. Литералы регулярных выражений могут присутствовать в операциях присваивания (после знака «равно»), в операциях проверки на равенство или неравенство (после знака «равно»), в операции отрицания (после восклицательного знака), в литералах определений объектов (после двоеточия) и в параметрах функций (после открывающей скобки или запятой). Пробельные символы между литералом регулярного выражения и предшествующим символом должны игнорироваться.

Решение

```
(?<=[=:(,](?:\s*!)?\s*)/[^/\\\r\n]*(?:\\.[^/\\\r\n]*)*/
```

Параметры: нет
Диалект: .NET

```
[=:(,](?:\s*!)?\s*\K/[^/\\\r\n]*(?:\\.[^/\\\r\n]*)*/
```

Параметры: нет
Диалекты: PCRE 7.2, Perl 5.10

```
(?<=[=:(,](?:\s{0,10}+!)?\s{0,10})/[^/\\\r\n]*(?:\\.[^/\\\r\n]*)*/
```

Параметры: нет
Диалекты: .NET, Java

```
[=:(,](?:\s*!)?+\s*(/[^/\\\r\n]*(?:\\.[^/\\\r\n]*)*/)
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Во всех четырех решениях для сопоставления с литералом регулярного выражения используется выражение ‹/[^/\\\r\n]*(?:\\.[^/\\\r\n]*)*/›. Это то же самое регулярное выражение, что было предложено в разделе «Решение» в рецепте 7.9, только вместо кавычек здесь используются символы слэша. В действительности, литерал регулярного выражения – это всего лишь строка, заключенная в символы прямого слэша, которая может содержать экранированные символы слэша.

Различия между этими четырьмя решениями заключаются в способе проверки предшествования знака «равно», двоеточия, открывающей круглой скобки или запятой и, возможно, присутствия восклицательного знака между этими символами и литералом регулярного выражения. Эту проверку можно было бы легко реализовать с помощью ретроспективной проверки, если бы по условиям задачи не требовалось игнорировать произвольное число пробельных символов между литералом регулярного выражения и предшествующим ему символом. Это требование усложняет решение, так как диалекты регулярных выражений, обсуждаемые в этой книге, существенно отличаются в поддержке ретроспективных проверок.

Диалект .NET – единственный из рассматриваемых здесь, который позволяет бесконечное повторение внутри ретроспективной проверки. Поэтому для .NET получилось замечательное решение: ‹(?<=[=:(,](?:\s*!)?\s*)›. Символьный класс ‹[=:(,]› проверяет присутствие любого из четырех символов. Подвыражение ‹(?:\s*!)?› допускает присутствие восклицательного знака после символа и произвольное число пробельных символов между ними. Следующее подвыражение ‹\s*› допускает присутствие произвольного числа пробельных символов перед символом слэша, открывающим литерал регулярного выражения.

Диалекты Perl и PCRE не допускают наличие квантификаторов в ретроспективных проверках. Поэтому решение на основе ретроспективной проверки в диалектах Perl и PCRE получается недостаточно гибким. Но в Perl 5.10 и PCRE 7.2 появился новый метасимвол `\K`, который можно использовать вместо ретроспективной проверки. Подвыражение `[=:(,](?:\s*!)?\s*` совпадает с любым из четырех символов, а также с любым числом последующих пробельных символов, восклицательным знаком и снова с любым числом последующих пробельных символов. После того как совпадет это подвыражение, метасимвол `\K` сообщит механизму регулярных выражений, что он должен сохранить только что найденное совпадение. Знаки пунктуации, только что найденные регулярным выражением, не будут включены в общий результат сопоставления. Затем процесс продолжается как обычно и выполняется попытка сопоставления `/[^/\\\r\n]*(?:\\.[^/\\\r\n]*)*/` с предполагаемым литералом регулярного выражения.

Диалект Java не поддерживает бесконечное повторение в ретроспективных проверках, но поддерживает ограниченное повторение. Поэтому вместо подвыражения `\s*`, проверяющего наличие неограниченного числа пробельных символов, используется подвыражение `\s{0,10}`, ограничивающееся 10 пробельными символами. Число 10 выбрано произвольно; число должно быть достаточно большим, чтобы гарантировать, что выражение не пропустит далеко отстоящий литерал регулярного выражения, и при этом достаточно маленьким, чтобы избежать неоправданного снижения скорости работы регулярного выражения. Чем большее число повторений будет позволено, тем больше символов придется проверить механизму регулярных выражений Java в поисках совпадения с содержимым ретроспективной проверки.

Остальные диалекты регулярных выражений либо не поддерживают квантификаторы внутри ретроспективных проверок, либо не поддерживают сами ретроспективные проверки, либо не поддерживают метасимвол `\K`. В решениях для этих диалектов просто используется подвыражение `[=:(,](?:\s*!)?+\s*`, совпадающее со знаками пунктуации, допустимыми перед литералом регулярного выражения, и подвыражение `(/[^/\\\r\n]*(?:\\.[^/\\\r\n]*)*/)`, совпадающее с литералом регулярного выражения и запоминающее его в сохраняющей группе. Общее совпадение с регулярным выражением будет включать и знак пунктуации, и литерал регулярного выражения. Применение сохраняющей группы упрощает извлечение литерала. Это решение будет работоспособным, только если приложение, в котором предполагается его использовать, позволяет извлекать совпадение не только со всем регулярным выражением, но и с сохраняющей группой.

См. также

Рецепт 2.16, где во всех подробностях описываются ретроспективные проверки и метасимвол `\K`.

7.11. Встроенные документы

Задача

Необходимо написать регулярное выражение, совпадающее со *встроенными документами* (*here documents*) в файлах с исходным программным кодом на языке сценариев, в котором встроенные документы могут начинаться с последовательности << и следующим за ней некоторым словом. Это слово может быть заключено в апострофы или кавычки. Конец встроенного документа обозначается этим же словом, находящимся в самом начале текстовой строки, без кавычек, с сохранением регистра символов в нем.

Решение

```
<<(["']?)([A-Za-z]+)\b\1.*?^\2\b
```

Параметры: точке соответствуют границы строк, символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
<<(["']?)([A-Za-z]+)\b\1[\s\S]*?^\2\b
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Это регулярное выражение может показаться несколько замысловатым, но в действительности это не так. Подвыражению ‹<<› соответствует пара символов <<. Следующему подвыражению ‹(["']?)› соответствует необязательный апостроф или кавычка. Круглые скобки образуют сохраняющую группу для запоминания типа открывающей кавычки или ее отсутствия. Важно отметить, что квантификатор ‹?› должен находиться внутри группы, а не за ее пределами, чтобы группа всегда участвовала в сопоставлении. Если сделать необязательной саму группу, она не будет принимать участие в сопоставлении в отсутствие кавычки и попытка сослаться на нее с помощью обратной ссылки будет вызывать ошибку сопоставления.

Сохраняющая группа с символьным классом ‹([A-Za-z]+)› соответствует слову и делает его доступным для второй обратной ссылки. Проверка границы слова ‹\b› гарантирует совпадение с целым словом, следующим за ‹<<›. Если опустить ее, механизм регулярных выражений сможет выполнять возвраты и разрешить совпадение с частью этого слова, если сопоставление с обратной ссылкой ‹\2› потерпит неудачу. Проверять границу слова перед самим словом не требуется, потому что подвыражение ‹<<(["']?)› уже гарантирует наличие символа, не являющегося символом слова, перед самим словом.

‹\1› – это обратная ссылка на первую сохраняющую группу. Эта группа хранит кавычку, если она была найдена, или пустую строку. То есть ‹\1› совпадет с кавычкой того же типа, что и открывающая кавычка, сохраненная группой. ‹\1› не оказывает никакого влияния на работу регулярного выражения, если сохраняющая группа хранит пустую строку.

‹.*?› соответствует тексту любой длины. Требование к включению параметра «точке соответствуют границы строк» обеспечивает поддержку многострочного текста. Диалект JavaScript не поддерживает этот параметр, поэтому в решении для JavaScript используется подвыражение ‹[\s\S]*?›. В любом случае, знак вопроса превращает звездочку в минимальный квантификатор, требуя от нее совпадения с как можно меньшим количеством символов. Встроенный документ должен завершаться первым вхождением заключительного слова, а не последним. В файле может иметься несколько встроенных документов, ограниченных одним и тем же завершающим словом, и минимальный квантификатор обеспечит совпадение с каждым из этих документов по отдельности.

‹^› соответствует началу произвольной строки, потому что включен параметр «символам ^ и $ соответствуют границы строк». Диалект Ruby не поддерживает этот параметр, но так как в Ruby символ крышки и знак доллара всегда соответствуют границам строк, для этого диалекта не требуется создавать отдельное решение. Просто потребуется устанавливать на один параметр меньше.

‹\2› – это обратная ссылка на вторую сохраняющую группу. Эта группа хранит слово, найденное в начале встроенного документа. Так как синтаксис встроенных документов в предполагаемом языке сценариев различает регистр символов, регулярное выражение также должно быть чувствительно к регистру символов. Именно по этой причине для сопоставления со словом используется ‹[A-Za-z]+›, а не ‹[a-z]+› или ‹[A-Z]+› с включенным параметром нечувствительности к регистру. Кроме того, обратные ссылки сохраняют чувствительность к регистру символов даже при включенном параметре нечувствительности к регистру.

Наконец, еще одна проверка границы слова ‹\b› гарантирует прекращение процесса сопоставления, только если ‹\2› совпадет с искомым словом, а не с частью более длинного слова. Проверять границу перед словом не требуется, так как символ крышки уже гарантирует, что слово находится в начале строки. Всякий раз когда сопоставление с ‹\2› или заключительной границей слова ‹\b› потерпит неудачу, механизм регулярных выражений выполнит возврат и позволит подвыражению ‹.*?› совпасть с большим количеством символов.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6

говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.13 описывается, как обеспечить совпадение с как можно меньшим количеством символов.

7.12. Обобщенный формат файла журнала

Задача

Необходимо написать регулярное выражение, совпадающее с каждой строкой в файле журнала веб-сервера, имеющем обобщенный формат файлов журналов (Common Log Format).[1] Например:

```
127.0.0.1 - jg [27/Apr/2012:11:27:36 +0700] "GET /regexcookbook.html
HTTP/1.1" 200 2326
```

Регулярное выражение должно иметь сохраняющие группы для всех полей, чтобы приложение имело простую возможность обрабатывать записи по полям.

Решение

```
^(?<client>\S+)•\S+•(?<userid>\S+)•\[(?<datetime>[^\]]+)\]↵
•"(?<method>[A-Z]+)•(?<request>[^•"]+)?•HTTP/[0-9.]+"↵
•(?<status>[0-9]{3})•(?<size>[0-9]+|-)
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?P<client>\S+)•\S+•(?P<userid>\S+)•\[(?P<datetime>[^\]]+)\]↵
•"(?P<method>[A-Z]+)•(?P<request>[^•"]+)?•HTTP/[0-9.]+"↵
•(?P<status>[0-9]{3})•(?P<size>[0-9]+|-)
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: PCRE 4, Perl 5.10, Python

```
^(\S+)•\S+•(\S+)•\[([^\]]+)\]•"([A-Z]+)•([^•"]+)?•HTTP/[0-9.]+"↵
•([0-9]{3})•([0-9]+|-)•"([^"]*)"•"([^"]*)"
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Создание регулярных выражений, совпадающих с целой записью в файле журнала, – задача достаточно простая. Однако это утверждение верно, только когда все записи следуют одному и тому же формату, отличаясь только значениями полей. Данного правила придерживаются вебсерверы, ведущие журнал доступа с использованием обобщенного формата Common Log Format, такие как Apache. Каждая строка в файле

---

[1] *http://httpd.apache.org/docs/current/logs.html.*

журнала соответствует одной записи, а каждая запись состоит из семи полей, разделенных пробелами:

1. IP-адрес или имя хоста клиента, выполнившего запрос.
2. Идентификатор клиента согласно RFC 1413. Редко используется. Дефис в этом поле свидетельствует об отсутствии идентификатора.
3. Имя пользователя при использовании механизма HTTP-аутентификации или дефис, если механизм HTTP-аутентификации не использовался.
4. Время приема запроса в квадратных скобках. Обычно в формате [день/месяц/год:часы:минуты:секунды часовой_пояс], причем время в 24-часовом формате.
5. Запрос, в кавычках, с тремя блоками информации, разделенными пробелами:
   a. Метод запроса[1], такой как `GET`, `POST` или `HEAD`.
   b. Запрошенный ресурс – часть строки URL, следующая за именем хоста.
   c. Версия протокола – либо HTTP/1.0, либо HTTP/1.1.
6. Код состояния[2], трехзначное число, такое как `200` (означает «OK») или `404` («не найдено»).
7. Объем данных, отправленных клиенту, исключая заголовки. Если ответ не возвращался, в этом поле может быть дефис или ноль.

В действительности совсем необязательно знать все эти тонкости, чтобы написать регулярное выражение, успешно совпадающее с каждой записью. Можно уверенно предположить, что веб-сервер будет записывать в файл журнала только допустимую информацию. Регулярному выражению не требуется фильтровать журнал в поисках записей с определенными значениями, потому что эту функцию берет на себя приложение, использующее данное регулярное выражение.

Итак, фактически достаточно лишь знать, как отделяются поля и записи друг от друга. Обладая такими знаниями, можно будет организовать сопоставление с полями в виде отдельных сохраняющих групп. Записи отделяются разрывами строк, а поля – пробелами. Но поля с датой и запросом могут содержать пробелы, поэтому при реализации сопоставления с этими двумя полями необходимо проявить чуть больше внимания. Первые три поля не могут содержать пробелы. Благодаря этому совпадения с ними можно описать с использованием сокращения символьного класса `‹\S+›`, что соответствует последовательности из одного или более символов, не содержащей пробелов или границ строк. Так как идентификатор клиента редко используется на практике, совпадение с ним не сохраняется.

---

[1] *http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.*

[2] *http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.*

Поле с датой всегда заключено в квадратные скобки, которые являются метасимволами регулярных выражений. Чтобы обеспечить совпадение с квадратными скобками, они экранируются: ‹\[› и ‹\]›. Строго говоря, закрывающую квадратную скобку можно не экранировать за пределами символьных классов. Но поскольку между литералами квадратных скобок находится символьный класс, экранирование закрывающей квадратной скобки упростит чтение регулярного выражения. Инвертированный символьный класс ‹[^\]]+› соответствует одному или более символам, не являющимся закрывающей квадратной скобкой. В JavaScript литерал закрывающей квадратной скобки должен экранироваться всегда, если он находится внутри символьного класса. Другие диалекты не требуют экранирования закрывающей квадратной скобки, если она следует непосредственно за открывающей квадратной скобкой или инвертирующим символом крышки, но в предложенных решениях она экранируется для большей ясности. Инвертированный символьный класс заключен в литералы квадратных скобок: ‹\[([^\]]+)\]›. Как следствие регулярное выражение будет сохранять дату без квадратных скобок, поэтому приложению, использующему его, не придется удалять скобки перед анализом даты.

Так как запрос фактически содержит три блока информации, для сопоставления с ним применяются три отдельные сохраняющие группы. ‹[A-Z]+› соответствует любому слову, состоящему только из символов верхнего регистра, – методу запроса. Название запрашиваемого ресурса может быть практически любой комбинацией символов. ‹[^ "]+› соответствует такой комбинации, исключая пробелы и кавычки. ‹HTTP/[0-9.]+› соответствует версии HTTP и допускает присутствие цифр и точек в любых комбинациях в номере версии.

Код состояния состоит из трех цифр, поэтому совпадение с ним легко можно описать подвыражением ‹[0-9]{3}›. Объем данных – это число или дефис, поэтому сопоставление выполняется с помощью подвыражения ‹[0-9]+|-›. Об объединении этих двух альтернатив заботится сохраняющая группа.

В начале регулярного выражения присутствует символ крышки, а параметр «символам ^ и $ соответствуют границы строк» гарантирует, что он будет совпадать только с позицией после границы строки. Благодаря этому сопоставление с каждой новой записью в файле журнала будет начинаться с начала строки, что существенно увеличивает производительность регулярного выражения при условии невысокой вероятности появления в файле недопустимых строк. Регулярное выражение будет пытаться выполнить сопоставление с такими строками только один раз, в начале строки, не проверяя каждую позицию в этой строке.

Мы не поместили знак доллара в конец регулярного выражения специально, чтобы ограничить длину записи длиной одной текстовой строки в файле. Если запись в файле журнала содержит дополнительную информацию, регулярное выражение попросту будет игнорировать ее. Это позволяет данному регулярному выражению работать одинаково хоро-

шо и с расширенными форматами файлов журналов, такими как Combined Log Format (комбинированный формат файлов журналов), о котором рассказывается в следующем рецепте.

Регулярное выражение состоит из восьми сохраняющих групп. Чтобы упростить его чтение, мы использовали именованные сохранения – в версии для диалектов, поддерживающих такую возможность. JavaScript (без XRegExp) и Ruby 1.8 – единственные диалекты из обсуждаемых в этой книге, не поддерживающие именованные сохранения. В версии для них используются нумерованные сохраняющие группы.

Варианты

```
^(?<client>\S+) \S+ (?<userid>\S+) \[(?<day>[0-9]{2})/(?<month>
[A-Za-z]+)/(?<year>[0-9]{4}):(?<hour>[0-9]{2}):(?<min>[0-9]{2}):
(?<sec>[0-9]{2}) (?<zone>[-+][0-9]{4})\] "(?<method>[A-Z]+) 
(?<file>[^#? "]+)(?<parameters>[#?][^ "]*)? HTTP/[0-9.]+" 
(?<status>[0-9]{3}) (?<size>[0-9]+|-)
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?P<client>\S+) \S+ (?P<userid>\S+) \[(?P<day>[0-9]{2})/(?P<month>
[A-Za-z]+)/(?P<year>[0-9]{4}):(?P<hour>[0-9]{2}):(?P<min>[0-9]{2}):
(?P<sec>[0-9]{2}) (?P<zone>[-+][0-9]{4})\] "(?P<method>[A-Z]+) 
(?P<file>[^#? "]+)(?P<parameters>[#?][^ "]*)? HTTP/[0-9.]+" 
(?P<status>[0-9]{3}) (?P<size>[0-9]+|-)
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: PCRE 4, Perl 5.10, Python

```
^(\S+) \S+ (\S+) \[([0-9]{2})/([A-Za-z]+)/([0-9]{4}):([0-9]{2}):
([0-9]{2}):([0-9]{2}) ([\-+][0-9]{4})\] "([A-Z]+) ([^#? "]+)
([#?][^ "]*)? HTTP/[0-9.]+" ([0-9]{3}) ([0-9]+|-)
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Регулярное выражение, представленное как решение в этом рецепте, просто возвращает совпадения со всеми полями, оставляя дальнейшую их обработку за приложением, использующим его. В зависимости от потребностей приложения может пригодиться возможность извлечения с помощью регулярного выражения некоторой дополнительной информации из записей в журнале.

В данном варианте выполняется сопоставление со всеми элементами даты и времени в отдельности, что может упростить преобразование совпавшего текста в фактическую дату и время внутри приложения. Кроме того, информация о запрошенном ресурсе разбивается на две части – «файл» и «параметры». Если запрошенный ресурс содержит символ ? или #, именованная группа `file` сохранит текст, предшествующий ? или #. Именованная группа `parameters` сохранит сам символ ? или #, а также все, что следует за ним. Это может, например, упростить подсчет страниц в приложении без учета параметров.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы, такие как символ крышки. В рецепте 2.11 обсуждаются именованные сохраняющие группы.

В главе 3 демонстрируются фрагменты программного кода, которые можно использовать в комплексе с этим регулярным выражением для реализации обработки файлов журналов в своих приложениях. В рецепте 3.11 приводится пример программного кода, выполняющего обход всех найденных совпадений, который пригодится в случае, когда приложение загружает весь файл журнала в строковую переменную. В рецепте 3.7 демонстрируется прием сопоставления с отдельными строками текста, когда приложение читает содержимое файла построчно. В рецепте 3.9 демонстрируется код, извлекающий фрагменты текста, совпавшие с сохраняющими группами.

7.13. Комбинированный формат файлов журналов

Задача

Необходимо написать регулярное выражение, совпадающее с каждой строкой в файле журнала веб-сервера, имеющем комбинированный формат файлов журналов (Combined Log Format).[1] Например:

```
127.0.0.1 - jg [27/Apr/2012:11:27:36 +0700] "GET /regexcookbook.html HTTP/1.1"
200 2326 "http://www.regexcookbook.com/" "Mozilla/5.0 (compatible; MSIE 9.0;
Windows NT 6.1; Trident/5.0)"
```

Решение

```
^(?<client>\S+)•\S+•(?<userid>\S+)•\[(?<datetime>[^\]]+)\]↵
•"(?<method>[A-Z]+)•(?<request>[^•"]+)?•HTTP/[0-9.]+"↵
•(?<status>[0-9]{3})•(?<size>[0-9]+|-)•"(?<referrer>[^"]*)"↵
•"(?<useragent>[^"]*)"
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?P<client>\S+)•\S+•(?P<userid>\S+)•\[(?P<datetime>[^\]]+)\]↵
•"(?P<method>[A-Z]+)•(?P<request>[^•"]+)?•HTTP/[0-9.]+"↵
•(?P<status>[0-9]{3})•(?P<size>[0-9]+|-)•"(?P<referrer>[^"]*)"↵
•"(?P<useragent>[^"]*)"
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: PCRE 4, Perl 5.10, Python

---

1 *http://httpd.apache.org/docs/current/logs.html.*

```
^(\S+)•\S+•(\S+)•\[([^\]]+)\]•"([A-Z]+)•([^•"]+)?•HTTP/[0-9.]+"↵
•([0-9]{3})•([0-9]+|-)•"([^"]*)"•"([^"]*)"•"([^"]*)"
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Комбинированный формат файлов журналов (Combined Log Format) имеет практически ту же структуру, что и обобщенный формат файлов журналов (Common Log Format), кроме двух дополнительных полей в конце каждой записи. Первое из этих двух полей содержит адрес URL веб-страницы, сославшейся на данный ресурс, а второе содержит название пользовательского программного агента (user agent). Значения обоих полей заключены в кавычки. Совпадение с каждым из этих полей обеспечивается с помощью простого регулярного выражения ‹"[^"]*"›. Мы заключили ‹[^"]*› в сохраняющую группу, чтобы упростить извлечение адреса URL веб-страницы, сославшейся на данный ресурс, и названия пользовательского программного агента без окружающих кавычек.

См. также

Предыдущий рецепт, где рассказывается, как реализовать совпадение с каждой записью в файле журнала веб-сервера, имеющем обобщенный формат файлов журналов.

7.14. Сообщения о недействительных ссылках в файле журнала веб-сервера

Задача

Имеется файл журнала веб-сервера в комбинированном формате файлов журналов (Combined Log Format). Необходимо найти в нем все сообщения о недействительных ссылках, имеющихся на веб-сайте.

Решение

```
"(?:GET|POST)•(?<file>[^#?•"]+)(?:[#?][^•"]*)?•HTTP/[0-9.]+"•404•↵
(?:[0-9]+|-)•"(?<referrer>http://www\.yoursite\.com[^"]*)"
```

Параметры: нет
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

```
"(?:GET|POST)•(?P<file>[^#?•"]+)(?:[#?][^•"]*)?•HTTP/[0-9.]+"•404•↵
(?:[0-9]+|-)•"(?P<referrer>http://www\.yoursite\.com[^"]*)"
```

Параметры: нет
Диалекты: PCRE 4, Perl 5.10, Python

```
"(?:GET|POST) ([^#? "]+)(?:[#?][^ "]*)? HTTP/[0-9.]+" 404 ↵
(?:[0-9]+|-) "(http://www\.yoursite\.com[^"]*)"
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Если пользователь щелкнет мышью на ссылке, указывающей на отсутствующий файл на этом же веб-сайте, он получит в ответ ошибку «страница не найдена». В поле адреса URL веб-страницы, сославшейся на данный ресурс, веб-сервер создаст запись в файле журнала, содержащую имя отсутствующего файла в поле запрошенного ресурса, код состояния 404 и страницу, содержащую недействительную ссылку. Вам необходимо извлечь поле запрошенного ресурса из записей в журнале, имеющих код состояния 404, и адрес URL веб-страницы, сославшейся на данный ресурс, принадлежащий вашему веб-сайту.

Один из способов получить желаемый результат – написать на одном из языков программирования сценарий, использующий регулярное выражение из рецепта 7.13, который будет выполнять обход всех совпадений, проверять, содержит ли группа `status` код `404` и начинается ли группа `referrer` с `http://www.yoursite.com`, и при выполнении обоих условий выводить текст совпадений с группами `referrer` и `request`. В целом это неплохое решение. Его преимущество заключается в возможности добавить в сценарий дополнительные проверки.

Однако поставленную в этом рецепте задачу легко решить с помощью одного только регулярного выражения без применения программного кода. Вы сможете открыть файл журнала в том же текстовом редакторе, в котором правите свой веб-сайт, и с помощью регулярного выражения, представленного как решение, отыскать ошибки 404, соответствующие недействительным ссылкам на вашем веб-сайте. Это регулярное выражение основано на выражении из рецепта 7.13. Мы рассмотрим процесс создания выражения на примере версии, использующей именованные сохранения в стиле .NET. Версии, использующие именованные сохранения в стиле Python или нумерованные сохранения, имеют ту же структуру – они отличаются только синтаксисом определения сохраняющих групп.

Фактически требуется только проверить, не содержит ли совпадение с группой `status` код 404 и не начинается ли совпадение с группой `referrer` с доменного имени вашего сайта:

```
^(?<client>\S+) \S+ (?<userid>\S+) \[(?<datetime>[^\]]+)\] ↵
 "(?<method>[A-Z]+) (?<request>[^ "]+)? HTTP/[0-9.]+" (?<status>404) ↵
 (?<size>[0-9]+|-) "(?<referrer>http://www\.yoursite\.com[^"]*)" ↵
 "(?<useragent>[^"]*)"
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Это регулярное выражение уже решает поставленную задачу. Но оно недостаточно эффективно. Выражение совпадает с целой записью в файле журнала, а нам требуются только группы request, status и referrer. Группа useragent вообще не влияет на совпадение, поэтому от нее вполне можно избавиться:

```
^(?<client>\S+)•\S+•(?<userid>\S+)•\[(?<datetime>[^\]]+)\]↵
•"(?<method>[A-Z]+)•(?<request>[^•"]+)?•HTTP/[0-9.]+"•(?<status>404)↵
•(?<size>[0-9]+|-)•"(?<referrer>http://www\.yoursite\.com[^"]*)"
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Убрать группы с client по method не так просто. Они привязывают регулярное выражение к началу строки и гарантируют правильное позиционирование групп с request по referrer. Если потребуется удалить какие-либо группы, находящиеся в начале регулярного выражения, необходимо гарантировать совпадение регулярного выражения с требуемыми полями. В случае с файлами журналов это не такая большая проблема. Большая часть полей имеет уникальное содержимое, и наше регулярное выражение достаточно детализировано. Оно явно проверяет наличие квадратных скобок и кавычек в полях, где они должны быть, допускает наличие только чисел в числовых полях, явно проверяет совпадение с фиксированным текстом, таким как «HTTP», и т. д. Если бы мы поддались лени и для сопоставления со всеми полями использовали ‹\S+›, мы не смогли бы сократить регулярное выражение еще больше, так как ‹\S+› совпадает практически со всем что угодно.

Нам также нужно сохранить эффективность регулярного выражения. Символ крышки в начале гарантирует, что попытка сопоставления будет выполняться только с начала каждой строки. Если совпадение со строкой не будет найдено из-за того, что код состояния не равен 404 или адрес URL веб-страницы, сославшейся на данный ресурс, содержит другое доменное имя, регулярное выражение сразу же перейдет к следующей строке в журнале. Если бы нам потребовалось убрать все, что предшествует группе ‹(?<request>[^•"]+)?›, регулярное выражение начиналось бы с подвыражения ‹[^•"]+›. Механизм регулярных выражений вынужден был бы тогда сравнить каждый символ в файле с пробелом и кавычкой. На больших файлах это существенно замедлило бы работу регулярного выражения.

Отличной точкой усечения головы данного регулярного выражения является подвыражение ‹"(?<method>[A-Z]+)›. Чтобы еще больше повысить его эффективность, можно указать, что интерес представляют всего две альтернативы:

```
"(?<method>GET|POST)•(?<request>[^•"]+)?•HTTP/[0-9.]+"•(?<status>404)↵
•(?<size>[0-9]+|-)•"(?<referrer>http://www\.yoursite\.com[^"]*)"
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java 7, XRegExp, PCRE 7, Perl 5.10, Ruby 1.9

Эта версия регулярного выражения начинается с литерала кавычки. Обычно регулярные выражения, начинающиеся с литерального текста, оказываются весьма эффективными, потому что механизмы регулярных выражений часто используют дополнительные оптимизации для подобных случаев. Каждая запись в файле журнала имеет шесть кавычек, то есть будет выполнено не более шести попыток сопоставить это регулярное выражение с каждой записью, не содержащей код состояния 404. В пяти случаях из шести сопоставление будет терпеть неудачу немедленно, на операторе выбора `‹GET|POST›`. Шесть попыток сопоставления на каждую строку может показаться менее эффективным решением, чем одна попытка, но это не так — немедленный отказ на сопоставлении с `‹GET|POST›` будет выполняться быстрее, чем сопоставление с `‹^(?<client>\S+)•\S+•(?<userid>\S+)•\[(?<datetime>[^\]]+)\]•›`.

И заключительной оптимизацией является отказ от неиспользуемых сохраняющих групп. Некоторые из них можно удалить полностью. Другие, как например группу с оператором выбора, можно заменить простыми несохраняющими группами. В результате получается регулярное выражение, представленное в разделе «Решение».

В заключительной версии остались две сохраняющие группы: `file` и `referrer`. Когда это регулярное выражение будет использоваться в текстовом редакторе или с инструментом grep, способными извлекать текст, совпавший с сохраняющими группами, вы сможете настроить свой инструмент на выбор текста только из групп `file` и `referrer`. Это позволит получить список недействительных ссылок и страниц, где они находятся, очищенный от лишней информации.

См. также

Рецепт 7.12, где рассказывается, как с помощью регулярного выражения анализировать записи в файле журнала веб-сервера. В нем также приводятся ссылки на рецепты в главе 2 с описанием синтаксических конструкций регулярных выражений, используемых в этом рецепте.

8

URL, пути и адреса в Интернете

Помимо чисел, которые рассматривались в главе 6, еще одной важной темой, которая касается широкого круга программ, является поиск совпадений с различными путями и адресами, которые используются для поиска данных:

- Адреса URL, URN и связанные с ними строки
- Доменные имена
- IP-адреса
- Имена файлов и папок в операционной системе Windows

Формат записи адресов URL оказался настолько гибким и удобным, что был принят для обозначения адресов различных ресурсов, которые вообще не имеют отношения к Всемирной паутине (World Wide Web). По этой причине приемы синтаксического анализа текста с помощью регулярных выражений, которые приводятся в этой главе, будут весьма полезны в самых разных ситуациях.

8.1. Проверка адресов URL

Задача

Необходимо проверить, является ли заданный фрагмент текста адресом URL, допустимым в некоторой ситуации.

Решение

Допускает практически любые адреса URL:

```
^(https?|ftp|file)://.+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A(https?|ftp|file)://.+\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Требует доменное имя и не допускает наличия имени пользователя или пароля:

```
\A                          # Якорь
(https?|ftp)://             # Схема
[a-z0-9-]+(\.[a-z0-9-]+)+ # Домен
([/?].*)?                   # Путь и/или параметры
\Z                          # Якорь
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(https?|ftp)://[a-z0-9-]+(\.[a-z0-9-]+)+↵
([/?].+)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Требует доменное имя и не допускает наличия имени пользователя или пароля. Позволяет опускать схему (http или ftp), если она может быть определена из имени поддомена (www или ftp):

```
\A                              # Якорь
((https?|ftp)://|(www|ftp)\.) # Схема адресации или поддомен
[a-z0-9-]+(\.[a-z0-9-]+)+     # Домен
([/?].*)?                       # Путь и/или параметры
\Z                              # Якорь
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^((https?|ftp)://|(www|ftp)\.)[a-z0-9-]+(\.[a-z0-9-]+)+([/?].*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Требует доменное имя и путь, ведущий к файлу с изображением. Имя пользователя, пароль или параметры не допускаются:

```
\A                          # Якорь
(https?|ftp)://             # Схема адресации
[a-z0-9-]+(\.[a-z0-9-]+)+ # Домен
(/[\w-]+)*                  # Путь
/[\w-]+\.(gif|png|jpg)      # Файл
\Z                          # Якорь
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(https?|ftp)://[a-z0-9-]+(\.[a-z0-9-]+)+(/[\w-]+)*/[\w-]+\.(gif|png|jpg)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Невозможно создать регулярное выражение, совпадающее со всеми допустимыми адресами URL и не совпадающее с каким-нибудь недопустимым адресом URL. Причина состоит в том, что практически все что угодно может стать допустимым адресом URL в еще не изобретенной схеме адресации.

Проверка адреса URL полезна, только когда известен контекст, в котором этот адрес может быть признан допустимым. В этом случае можно ограничить круг допустимых адресов URL в соответствии со схемами адресации, поддерживаемыми используемым программным обеспечением. Все регулярные выражения в этом рецепте предназначены для проверки URL, используемых веб-браузерами. Эти адреса URL записываются в следующем виде:

```
scheme://user:password@domain.name:80/path/file.ext?param=value&param2↵
=value2#fragment
```

Все эти части на практике являются необязательными. В URL типа `file:` используется только путь. В URL `http:` обязательным является только доменное имя.

Решения, представленные в этом рецепте, реализуют общепринятые правила определения допустимости адресов URL, используемые большинством веб-браузеров и других приложений. Они не пытаются реализовать правила, определяемые документом RFC 3986, являющимся официальным стандартом URL. Если вам требуется решение, соответствующее положениям RFC 3986, обращайтесь к рецепту 8.7.

Первое регулярное выражение в этом решении проверяет, начинается ли адрес URL с названия одной из схем адресации, наиболее часто используемых веб-браузерами: `http`, `https`, `ftp` и `file`. Якорный метасимвол ‹^› привязывает регулярное выражение к началу текста (рецепт 2.5). Конструкция выбора (рецепт 2.8) используется для представления списка схем. ‹https?› – это интеллектуальный способ представить конструкцию ‹http|https›.

Поскольку первое регулярное выражение допускает достаточно разные схемы адресации, такие как `http` и `file`, оно не пытается проверить корректность текста, следующего за именем схемы. Выражение ‹.+$› просто поглощает все, что встретится ему до конца текста, при условии, что текст не содержит символов разрыва строки.

По умолчанию точке (рецепт 2.4) соответствуют все символы, за исключением символов разрыва строки. В этом отношении диалект Ruby является исключением. В Ruby символ крышки и знак доллара всегда

совпадают с символами разрыва строки, и поэтому вместо них мы использовали метасимволы ‹\A› и ‹\Z› (рецепт 2.5). Строго говоря, то же самое следовало бы сделать со всеми регулярными выражениями в этом рецепте, чтобы адаптировать их для работы с диалектом Ruby. Следовало бы, – если бы входной текст мог бы содержать несколько строк и требовалось бы избежать совпадения с URL, занимающим одну из строк в тексте.

Следующие два регулярных выражения записаны в режиме свободного форматирования (рецепт 2.18) и в обычном виде. Выражение, записанное в режиме свободного форматирования, проще читать, тогда как обычное регулярное выражение легче вводить. Диалект JavaScript не поддерживает режим свободного форматирования регулярных выражений.

Эти два регулярных выражения принимают только адреса URL для веб и FTP и требуют, чтобы за именем схемы HTTP или FTP следовало нечто, похожее на допустимое доменное имя. Доменное имя должно записываться символами ASCII. Интернационализированные домены (IDN) не принимаются. За доменом может следовать путь или список параметров, отделяемый от домена символом слэша или знаком вопроса. Поскольку знак вопроса находится внутри символьного класса (рецепт 2.3), его не требуется экранировать. В символьных классах знак вопроса интерпретируется как обычный символ, а символ слэша интерпретируется как обычный символ в любом месте регулярного выражения. (Если его и можно встретить в исходных текстах программ экранированным обратным слэшем, то только потому, что в языке Perl и некоторых других символ слэша используется как разделитель в литералах регулярных выражений.)

В выражении не предпринимается попыток проверить корректность пути или параметров. Элемент ‹.*› просто совпадает со всем подряд, кроме символов разрыва строки. Поскольку и путь, и параметры являются необязательными, подвыражение ‹[/?].*› заключено в группу, которая сделана необязательной с помощью квантификатора ‹?› (рецепт 2.12).

Эти регулярные выражения и те, что следуют ниже, не допускают возможности передачи имени пользователя или пароля в строке адреса URL. Размещение информации о пользователе в строке URL считается плохой практикой из соображений безопасности.

Большинство веб-браузеров принимают адреса URL, которые не содержат имени схемы адресации, и правильно определяют ее исходя из доменного имени. Например, адрес `www.regexbuddy.com` – это сокращенная форма записи адреса `http://www.regexbuddy.com`. Чтобы обеспечить возможность совпадения с такими адресами URL, мы просто расширили в регулярном выражении список допустимых схем, включив в него поддомены `www.` и `ftp.`.

Выражение ‹(https?|ftp)://|(www|ftp)\.› прекрасно справляется с этим. В этом списке присутствуют две альтернативы, каждая из которых начинается с двух других альтернатив. Первая группа альтернатив обес-

печивает совпадение с ‹https?› и ‹ftp›, за которыми должна следовать последовательность ‹://›. Вторая группа альтернатив обеспечивает совпадение с ‹www› и ‹ftp›, за которыми должна следовать точка. Вы легко можете отредактировать оба списка, чтобы изменить названия схем адресации и имена поддоменов.

Последние два регулярных выражения требуют наличия в URL схемы, доменного имени, записанного символами ASCII, пути и имени файла с изображением в формате GIF, PNG или JPEG. Путь и имя файла могут состоять из символов и цифр любого алфавита, а также включать символы подчеркивания и дефисы. Все это, кроме дефиса, включает сокращенная форма записи символьного класса ‹\w› (рецепт 2.3).

Какое из этих регулярных выражений следует использовать? Это во многом зависит от того, что вы собираетесь делать. Во многих ситуациях ответом на этот вопрос может быть отказ от использования регулярных выражений вообще. Вместо этого нужно будет просто попытаться выполнить запрос по проверяемому адресу. Если в ответ будет возвращено допустимое содержимое, этот адрес можно признать корректным. Если будет получена ошибка с кодом 404, адрес можно отвергнуть. В конечном счете, это единственный способ проверить действительность адреса URL.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

Рецепт 8.7, в котором описывается решение, совместимое с RFC 3986.

8.2. Поиск адресов URL в тексте

Задача

Требуется отыскать адреса URL в текстовом документе. Адреса URL могут окружаться такими знаками пунктуации, как круглые скобки, не являющиеся частью адреса.

Решение

Адреса URL без пробелов:

```
\b(https?|ftp|file)://\S+
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Адреса URL без пробелов или завершающих знаков пунктуации:

```
\b(https?|ftp|file)://[-A-Z0-9+&@#/%?=~_|$!:,.;]*↵
[A-Z0-9+&@#/%=~_|$]
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Адреса URL без пробелов или завершающих знаков пунктуации. Наименование схемы адресации может отсутствовать в адресах, начинающихся с имени поддомена `www` или `ftp`:

```
\b((https?|ftp|file)://|(www|ftp)\.)[-A-Z0-9+&@#/%?=~_|$!:,.;]*↵
[A-Z0-9+&@#/%=~_|$]
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Пусть имеется текст:

```
Visit http://www.somesite.com/page, where you will find more information.
```

Какая его часть является адресом URL?

Прежде чем указать на фрагмент `http://www.somesite.com/page`, подумайте о следующем: знаки пунктуации и пробелы являются допустимыми символами в адресах URL. Хотя документ RFC 3986 (рецепт 8.7) не допускает появление литералов пробелов в адресах URL, все основные браузеры принимают и прекрасно справляются с пробелами в URL. Некоторые визуальные средства создания веб-страниц позволяют пользователям вставлять пробелы в имена файлов и папок и включают эти пробелы в виде литералов в ссылки на эти файлы.

Это означает, что регулярное выражение, совпадающее со всеми допустимыми адресами URL, в предыдущем тексте должно совпасть с фрагментом:

```
http://www.somesite.com/page, where you will find more information.
```

Вероятность, что человек, напечатавший это предложение, предполагал включить пробелы в адрес URL, весьма невелика. Первое регулярное выражение в решении исключает их с помощью сокращенной формы записи символьного класса ‹\S›, который совпадает со всеми символами, не являющимися пробельными. Даже при том, что предусматривается работа регулярного выражения в режиме «нечувствительности к регистру символов», символ S должен быть символом верхнего регистра, потому что метасимвол ‹\S› – это далеко не то же самое, что метасимвол ‹\s›. Они имеют совершенно противоположные значения. Подробнее о них рассказывается в рецепте 2.3.

Первое регулярное выражение все еще достаточно сырое. Оно включит в совпадение запятую, которая следует за адресом URL в примере. Не-

смотря на то что запятые и другие знаки пунктуации не являются чем-то необычным в адресах URL, тем не менее в конце адреса они используются крайне редко.

В следующем регулярном выражении вместо одного метасимвола `\S` используются два символьных класса. Первый класс включает в себя большее число знаков пунктуации, чем второй. Второй класс содержит знаки пунктуации, которые в соответствии с правилами английского языка с наибольшей вероятностью могут появиться в предложении после адреса URL. Первый класс имеет квантификатор «звездочка» (рецепт 2.12), который обеспечивает возможность совпадения с URL произвольной длины. Второй символьный класс не имеет квантификатора, требуя тем самым, чтобы адрес URL оканчивался одним из символов, перечисленных в этом классе. Символьный класс не содержит алфавитных символов нижнего регистра, потому что предполагается работа в режиме «нечувствительности к регистру символов». Порядок установки режимов в различных языках программирования описывается в рецепте 3.4.

Второе регулярное выражение будет неправильно определять адреса URL, в которых некорректно используются знаки пунктуации, совпадая лишь с частью адреса. Но это регулярное выражение решает наиболее типичную проблему, связанную с запятой или точкой, следующей сразу за адресом URL.

Большинство веб-браузеров принимают адреса URL, в которых отсутствует указание схемы адресации, корректно определяя ее исходя из доменного имени. Например, адрес `www.regexbuddy.com` является сокращенной формой записи адреса `http://www.regexbuddy.com`. Чтобы обеспечить возможность совпадения с такими адресами URL, последнее регулярное выражение расширяет список допустимых схем адресации, включая в него поддомены `www.` и `ftp.`.

Подвыражение `(https?|ftp)://|(www|ftp)\.` прекрасно справляется с этой задачей. В этом списке присутствуют две альтернативы, каждая из которых в свою очередь начинается с двух альтернатив. Первая альтернатива совпадает с `https?` и `ftp`, за которыми следует последовательность символов `://`. Вторая альтернатива совпадает с `www` или `ftp`, за которыми следует точка. Список схем и поддоменов, принимаемых регулярным выражением, может быть изменен в соответствии с конкретной задачей.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

В рецепте 8.5 приводится замещающий текст, который можно использовать в комбинации с этим регулярным выражением для реализации операции поиска с заменой, выполняющей преобразование адресов URL в гиперссылки HTML.

8.3. Поиск в тексте адресов URL, заключенных в кавычки

Задача

Требуется отыскать адреса URL в текстовом документе. Адреса URL могут быть окружены знаками пунктуации, являющимися частью окружающего текста, а не самого адреса URL. Необходимо предоставить пользователю возможность заключать адреса URL в кавычки, чтобы можно было явно показать, когда пробелы и знаки пунктуации являются частью URL.

Решение

```
\b(?:(?:https?|ftp|file)://|(www|ftp)\.)[-A-Z0-9+&@#/%?=~_|$!:,.;]*
                                        [-A-Z0-9+&@#/%=~_|$]
|"(?:(?:https?|ftp|file)://|(www|ftp)\.)[^"\r\n]+"
|'(?:(?:https?|ftp|file)://|(www|ftp)\.)[^'\r\n]+'
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов, точке соответствуют границы строк, якорным метасимволам соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

В предыдущем рецепте рассматривалась задача поиска адресов URL в тексте и как отличать знаки пунктуации в тексте от символов URL. И хотя решение в предыдущем рецепте пригодно для использования в большинстве случаев, тем не менее ни одно регулярное выражение не может гарантировать абсолютную надежность во всех ситуациях.

Если предполагается, что регулярное выражение будет применяться к создаваемым впоследствии текстовым документам, то можно предоставить пользователям возможность заключать адреса URL в кавычки. Решение, представленное здесь, позволяет заключать адреса URL в одинарные или двойные кавычки. Если адрес URL помещен в кавычки, он должен начинаться с одной из схем `‹https?|ftp|file›` или с одного из поддоменов `‹www|ftp›`. Вслед за схемой или поддоменом регулярное выражение допускает наличие любых символов, за исключением разрывов строк и кавычек, окружающих URL.

Регулярное выражение делится на три альтернативы. Первая альтернатива — это регулярное выражение из предыдущего рецепта, совпадающее

с адресом URL без кавычек и пытающееся отличать знаки пунктуации и символы URL. Вторая альтернатива совпадает с адресом URL в кавычках. Третья альтернатива совпадает с адресом URL в апострофах. Мы использовали две альтернативы вместо одной – с сохраняющей группой, включающей открывающую кавычку, и обратной ссылкой на закрывающую, – потому что обратную ссылку нельзя использовать внутри инвертированного символьного класса, исключающего кавычки из URL.

Мы использовали кавычки и апострофы, потому что таким образом обычно оформляются адреса URL в файлах HTML и XHTML. Заключение адресов в кавычки привычно для тех, кто занимается разработкой веб-страниц, но вы легко сможете добавить в регулярное выражение другие пары символов, ограничивающих адреса URL.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

8.4. Поиск в тексте адресов URL, заключенных в скобки

Задача

Требуется отыскать адреса URL в текстовом документе. Адреса URL могут окружаться знаками пунктуации, являющимися частью окружающего текста, а не самого адреса URL. Необходимо обеспечить корректное совпадение с адресами URL, содержащими пары круглых скобок, при этом не включая в совпадение скобки, окружающие адрес URL.

Решение

```
\b(?:(?:https?|ftp|file)://|www\.|ftp\.)
  (?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[-A-Z0-9+&@#/%=~_|$?!:,.])*
  (?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[A-Z0-9+&@#/%=~_|$])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\b(?:(?:https?|ftp|file)://|www\.|ftp\.)(?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)↵
|[-A-Z0-9+&@#/%=~_|$?!:,.])*(?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|↵
```

```
[A-Z0-9+&@#/%=~_|$])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

В адресах URL допустимым считается практически любой символ, включая и круглые скобки. Однако скобки очень редко встречаются в адресах, и именно поэтому мы не включили их в регулярные выражения, которые приводились в предыдущих рецептах. Тем не менее на некоторых крупных веб-сайтах скобки все же используются:

```
http://en.wikipedia.org/wiki/PC_Tools_(Central_Point_Software)
http://msdn.microsoft.com/en-us/library/aa752574(VS.85).aspx
```

Одно из решений заключается в том, чтобы потребовать от пользователя заключать такие адреса URL в кавычки. Другое – расширить регулярное выражение так, чтобы оно принимало такие URL. Самое сложное состоит в том, чтобы определить, какая из закрывающих скобок является частью URL, а какая является знаком пунктуации, завершающим URL, как в следующем примере:

```
RegexBuddy's website (at http://www.regexbuddy.com) is really cool.
```

Так как вполне возможно, что одна из скобок является ограничивающим знаком пунктуации, а другая – нет, мы не можем использовать прием заключения адреса в кавычки из предыдущего рецепта. Наиболее простое решение состоит в том, чтобы разрешить появление скобок внутри адресов URL, только когда они не являются вложенными парами открывающих и закрывающих скобок. Адреса URL Википедии и Microsoft отвечают этим требованиям.

Оба регулярных выражения, которые приводятся в решении, делают одно и то же. Первое из них записано в режиме свободного форматирования, чтобы упростить его чтение.

Эти регулярные выражения, по сути, представляют собой последнее регулярное выражение из рецепта 8.2. Они делятся на три части: список схем адресации, затем тело URL, в котором используется квантификатор «звездочка», чтобы обеспечить совпадение с URL любой длины, и завершающий символ URL, для которого не предусмотрен квантификатор (то есть этот символ должен появляться точно один раз). В оригинальном регулярном выражении в рецепте 8.2 вторая и третья части (тело URL и завершающий символ) состояли из единственного символьного класса.

В решениях для этого рецепта они были заменены двумя более сложными символьными классами. Символьный класс в середине:

```
[-A-Z0-9+&@#/%=~_|$?!:,.]
```

превратился в

```
\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[-A-Z0-9+&@#/%=~_|$?!:,.]
```

Последний символьный класс:

```
[A-Z0-9+&@#/%=~_|$]
```

превратился в

```
\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[A-Z0-9+&@#/%=~_|$]
```

Оба символьных класса заменили конструкции выбора (рецепт 2.8). Поскольку оператор выбора имеет самый низкий приоритет среди всех операторов регулярных выражений, для объединения альтернатив мы использовали несохраняющую группировку (рецепт 2.9).

В оба символьных класса была добавлена альтернатива ‹\([-A-Z0-9+&@#/%=~_|$?!:,.]* \)›, при этом оригинальный класс остался в качестве другой альтернативы. Новая альтернатива совпадает с парами круглых скобок с произвольным числом символов, допустимых в адресах URL, между ними.

Заключительный символьный класс был объединен с той альтернативой, которая позволяет адресам URL оканчиваться текстом в круглых скобках или единственным символом, который наименее вероятно может служить знаком пунктуации.

В результате объединения получилось регулярное выражение, совпадающее с адресами URL, включающими любое число круглых скобок, а также с адресами URL без круглых скобок и даже с адресами URL, не содержащими ничего, кроме скобок, при условии, что они парные.

В часть регулярного выражения, совпадающего с телом URL, мы добавили квантификатор «звездочка» для всей несохраняющей группы. Это обеспечило возможность появления любого числа пар круглых скобок в URL. Так как теперь вся несохраняющая группа получила квантификатор «звездочка», отпала необходимость в квантификаторе «звездочка» для оригинального символьного класса. На самом деле, звездочку необходимо убрать в обязательном порядке.

Средняя часть регулярного выражения, представленного в решении, имеет вид ‹(ab*c|d)*›, где ‹a› и ‹c› – это литералы круглых скобок, а ‹b› и ‹d› – символьные классы. Записать это регулярное выражение как ‹(ab*c|d*)*› будет ошибкой. На первый взгляд оно может выглядеть логично, потому что допускает совпадение с любым числом символов из класса ‹d›, но внешний квантификатор ‹*› уже обеспечивает повторение класса ‹d›. Если для класса ‹d› добавить еще одну звездочку, сложность регулярного выражения вырастет экспоненциально. Выражение ‹(d*)*› может совпасть со строкой dddd множеством способов. Например, внешняя звездочка может выполнить четыре повторения, а внутренняя звездочка по одному повторению каждый раз. Внешняя звездочка может выполнить три повторения, а внутренняя звездочка – 2-1-1, 1-2-1 или 1-1-2. Внешняя звездочка может выполнить два повторения, а внутренняя звездочка – 2-2, 1-3 или 3-1. Нетрудно понять, что с ростом длины

испытуемой строки количество возможных комбинаций увеличивается лавинообразно. Мы называем этот эффект катастрофическим возвратом; этот термин был введен в рецепте 2.15. Данная проблема проявляется, когда регулярное выражение не в состоянии отыскать допустимое совпадение, например, потому что в регулярное выражение было что-то добавлено, чтобы получить возможность находить адреса URL, после или внутри которых содержится что-то специфическое, отвечающее требованиям конкретной задачи.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

В рецепте 8.5 приводится замещающий текст, который можно использовать в комбинации с этим регулярным выражением для реализации операции поиска с заменой, выполняющей преобразование адресов URL в гиперссылки HTML.

8.5. Преобразование адресов URL в ссылки

Задача

Имеется текстовый документ, который может содержать один или более адресов URL. Необходимо преобразовать эти адреса в ссылки, добавив в текст HTML-теги ‹a›, заключив в них найденные URL. Сами адреса URL должны присутствовать в тексте документа как в виде адреса, на который указывает ссылка, так и в виде текста ссылки.

Решение

Чтобы отыскать адреса URL в тексте, можно воспользоваться одним из регулярных выражений из рецептов 8.2 или 8.4, а в качестве замещающего текста использовать:

```
<a•href="$&">$&</a>
```

Диалекты замещающего текста: .NET, JavaScript, Perl

```
<a•href="$0">$0</a>
```

Диалекты замещающего текста: .NET, Java, XRegExp, PHP

```
<a•href="\0">\0</a>
```

Диалекты замещающего текста: PHP, Ruby

```
<a href="\&">\&</a>
```

Диалект замещающего текста: Ruby

```
<a href="\g<0>">\g<0></a>
```

Диалект замещающего текста: Python

При работе над программой можно реализовать эту операцию поиска с заменой, как описано в рецепте 3.15.

Обсуждение

Эта задача решается достаточно просто. Задействуется регулярное выражение, совпадающее с адресом URL, и затем найденное совпадение замещается текстом «*URL*», где *URL* представляет найденный адрес URL. В различных языках программирования используется различный синтаксис определения замещающего текста; этим обусловлен такой длинный список решений для этой задачи. Но все они, по сути, делают одно и то же. Описание синтаксиса замещающего текста приводится в рецепте 2.20.

См. также

Рецепты 8.2 и 8.4, где описываются регулярные выражения, которые можно использовать в комбинации с этим замещающим текстом.

Приемы, использовавшиеся в замещающем тексте в этом рецепте, обсуждаются в главе 2. В рецепте 2.21 демонстрируется, как вставлять текст, совпавший с сохраняющей группой, в текст замены.

В рецепте 3.15 описывается, как можно реализовать операцию поиска с заменой в программном коде.

8.6. Проверка строк URN

Задача

Требуется проверить, является ли строка допустимым унифицированным именем ресурса (Uniform Resource Name, URN) в соответствии с определением в RFC 2141, или отыскать строки URN в текстовом документе.

Решение

Проверить, является ли строка допустимым унифицированным именем ресурса:

```
\Aurn:
# Идентификатор пространства имен
[a-z0-9][a-z0-9-]{0,31}:
# Строка из определенного пространства имен
```

```
[a-z0-9()+,\-.:=@;$_!*'%/?#]+
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^urn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Поиск строк URN в текстовом документе:

```
\burn:
# Идентификатор пространства имен
[a-z0-9][a-z0-9-]{0,31}:
# Строка из определенного пространства имен
[a-z0-9()+,\-.:=@;$_!*'%/?#]+
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\burn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]+
```

Парметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Поиск строки URN в текстовом документе; при этом предполагается, что знак пунктуации в конце URN является частью окружающего текста (английского), а не самой строки URN:

```
\burn:
# Идентификатор пространства имен
[a-z0-9][a-z0-9-]{0,31}:
# Строка из определенного пространства имен
[a-z0-9()+,\-.:=@;$_!*'%/?#]*[a-z0-9+=@$/]
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\burn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]*[a-z0-9+=@$/]
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Строка URN состоит из трех частей. Первая часть – это четыре символа `urn:`, которые можно добавить в регулярное выражение как литералы.

Вторая часть – это идентификатор пространства имен (Namespace Identifier, NID). Он занимает от 1 до 32 символов. Первый символ должен быть буквой или цифрой. Остальные символы могут быть буквами, цифрами или дефисами. Совпадение с идентификатором обеспечивается

с помощью двух символьных классов (рецепт 2.3): первому классу соответствует буква или цифра, а второму – от 0 до 31 букв, цифр и дефисов. Идентификатор должен отделяться от остальной части URN двоеточием, которое также можно добавить в регулярное выражение в виде литерала.

Третья часть URN – это строка из определенного пространства имен (Namespace Specific String, NSS). Она может иметь любую длину и в дополнение к буквам и цифрам может включать любые знаки пунктуации. Сопоставление с ней можно легко организовать с помощью другого символьного класса. Знак плюс, находящийся после символьного класса, повторяет его один или более раз (рецепт 2.12).

Если необходимо проверить, является ли заданная строка допустимой строкой URN, достаточно лишь добавить якорные метасимволы в начало и в конец регулярного выражения, чтобы обеспечить его привязку к началу и концу текста. Сделать это можно с помощью ‹^› и ‹$› во всех диалектах, за исключением Ruby, или с помощью ‹\A› и ‹\Z› во всех диалектах, за исключением JavaScript. Подробнее об этих якорных метасимволах рассказывается в рецепте 2.5.

Задача становится чуть сложнее, когда есть потребность отыскать строки URN в текстовом документе. Проблема знаков пунктуации, присущая адресам URL и обсуждавшаяся в рецепте 8.2, также свойственна и строкам URN. Предположим, что имеется текст:

```
The URN is urn:nid:nss, isn't it?
```

Проблема заключается в определении, является ли запятая частью URN. Строки URN, оканчивающиеся запятой, синтаксически являются допустимыми, но любой человек, читающий это предложение, поймет, что запятая является знаком препинания, а не частью URN. Последнее регулярное выражение в разделе «Решение» решает эту проблему, накладывая ограничение, не предусмотренное документом RFC 2141. Оно ограничивает последний символ строки URN набором символов, допустимых в части NSS, но маловероятных в качестве знаков препинания, которые могут появляться в предложении на английском языке, где упоминается эта строка URN.

Этого легко добиться, заменив квантификатор «плюс» (один или более раз) звездочкой (ноль или более раз) и добавив второй символьный класс, совпадающий с последним символом. Если бы мы добавили второй символьный класс, не изменив квантификатор, тем самым мы потребовали бы, чтобы часть NSS содержала как минимум два символа, а это не совсем то, что нам требуется.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.12

объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.7. Проверка универсальных адресов URL

Задача

Необходимо проверить, является ли заданный фрагмент текста адресом URL в соответствии с требованиями RFC 3986.

Решение

```
\A
(# Схема
 [a-z][a-z0-9+\-.]*:
 (# Авторизация и путь
  //
  ([a-z0-9\-._~%!$&'()*+,;=]+@)?                # Пользователь
  ([a-z0-9\-._~%]+                              # Имя хоста
  |\[[a-f0-9:.]+\]                              # IP-адрес хоста версии IPv6
  |\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей версии IPv...
  (:[0-9]+)?                                    # Порт
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?            # Путь
 |# Путь без авторизации
  (/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# Относительный URL (без схемы или авторизации)
 (# Относительный путь
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Абсолютный путь
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Строка запроса
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\A
(# Схема
 (?<scheme>[a-z][a-z0-9+\-.]*):
 (# Авторизация и путь
  //
  (?<user>[a-z0-9\-._~%!$&'()*+,;=]+@)?             # Пользователь
  (?<host>[a-z0-9\-._~%]+                           # Имя хоста
```

```
 |       \[[a-f0-9:.]+\]                                  # IP-адрес версии IPv6
 |       \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей
                                                        # версии IPv...
 (?<port>:[0-9]+)?                                      # Порт
 (?<path>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)            # Путь
|# Путь без авторизации
 (?<path>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
         (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# Относительный URL (без схемы или авторизации)
 (?<path>
  # Относительный путь
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Абсолютный путь
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Строка запроса
(?<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(?<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Perl 5.10, Ruby 1.9

```
\A
(# Схема
 (?<scheme>[a-z][a-z0-9+\-.]*):
 (# Авторизация и путь
  //
  (?<user>[a-z0-9\-._~%!$&'()*+,;=]+@)?                 # Пользователь
  (?<host>[a-z0-9\-._~%]+                               # Имя хоста
  |       \[[a-f0-9:.]+\]                               # IP-адрес версии IPv6
  |       \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей
                                                        # версии IPv...
  (?<port>:[0-9]+)?                                     # Порт
  (?<hostpath>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)   # Путь
 |# Путь без авторизации
  (?<schemepath>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
                (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# Относительный URL (без схемы или авторизации)
 (?<relpath>
  # Относительный путь
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Абсолютный путь
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
```

```
# Строка запроса
(?<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(?<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java 7, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(# Схема
 (?P<scheme>[a-z][a-z0-9+\-.]*):
 (# Авторизация и путь
  //
  (?P<user>[a-z0-9\-._~%!$&'()*+,;=]+@)?              # Пользователь
  (?P<host>[a-z0-9\-._~%]+                            # Имя хоста
  |        \[[a-f0-9:.]+\]                            # IP-адрес версии IPv6
  |        \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей
                                                      # версии IPv...й
  (?P<port>:[0-9]+)?                                  # Порт
  (?P<hostpath>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)    # Путь
 |# Путь без авторизации
  (?P<schemepath>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
                 (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# Относительный URL (без схемы или авторизации)
 (?P<relpath>
  # Относительный путь
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Абсолютный путь
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Строка запроса
(?P<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(?P<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

```
^([a-z][a-z0-9+\-.]*:(\/\/([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↵
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↵
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?|(\/?[a-z0-9\-._~%!$&'()*+,;=:@]+↵
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?)?)|([a-z0-9\-._~%!$&'()*+,;=@]+↵
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?|(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)↵
+\/?))
(\?[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?(#[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Большинство предыдущих рецептов в этой главе, имевших отношение к адресам URL, и регулярные выражения, приведенные в них, были рассчитаны на особые разновидности адресов URL. Некоторые из регулярных выражений были адаптированы для конкретных нужд, таких как определение, является ли знак пунктуации частью URL или текста, где приводится этот адрес URL.

Регулярные выражения в данном рецепте предусматривают работу с универсальными адресами URL. Они не ориентированы на поиск адресов URL в текстовых документах, а предназначены для проверки строк, которые, как предполагается, являются адресами URL, и деления адресов URL на отдельные части. Они решают все эти задачи для любых типов URL, при этом на практике часто требуется более тонкая настройка регулярных выражений. Рецепты, следующие далее, демонстрируют такие специфические регулярные выражения.

В документе RFC 3986 описывается, как должен выглядеть допустимый адрес URL. Он охватывает все возможные разновидности адресов URL, включая относительные URL и адреса URL для схем адресации, которые еще не используются. В результате документ RFC 3986 получился чрезвычайно обширным, а регулярные выражения, реализующие его требования, – очень длинными. Регулярные выражения в этом рецепте реализуют только самые основные требования. Они достаточно надежно разбивают URL на различные части, но не выполняют проверку каждой из этих частей. Проверка всех частей могла бы потребовать точных сведений о каждой схеме адресации, используемой в адресах URL.

Документ RFC 3986 охватывает не все адреса URL, которые могут встречаться на практике. Например, многие веб-серверы и веб-браузеры принимают адреса URL с литералами пробелов в них, хотя RFC 3986 требует, чтобы пробелы были представлены как `%20`.

Абсолютный адрес URL должен начинаться с названия схемы, такой как `http:` или `ftp:`. Первый символ в названии схемы должен быть буквой. Последующие символы могут быть буквами, цифрами и некоторыми знаками пунктуации. Соблюдение этого требования можно легко реализовать с помощью двух символьных классов: `‹[a-z][a-z0-9+\-.]*›`.

Во многих схемах адресации URL требуется то, что в RFC 3986 называется «авторизацией» («authority»). Авторизация – это доменное имя или IP-адрес сервера, которому может предшествовать имя пользователя и за которым может следовать номер порта.

Имя пользователя может состоять из букв, цифр и знаков пунктуации. Оно должно отделяться от доменного имени или IP-адреса знаком @. Подвыражение `‹[a-z0-9\-._~%!$&'()*+,;=]+@›` совпадает с именем пользователя и разделительным знаком.

Документ RFC 3986 достаточно либерально относится к набору символов, которые могут составлять доменное имя. В рецепте 8.15 описывает-

ся, какие символы обычно допустимы в доменных именах: буквы, цифры, дефисы и точки. Кроме того, в RFC 3986 допускается использовать тильду и любые другие символы в формате записи со знаком процента. Доменное имя должно преобразовываться в кодировку UTF-8, а любые байты, которые не являются буквами, цифрами, дефисами или тильдой, должны быть представлены в формате *%FF*, где *FF* – это шестнадцатеричное представление значения байта.

Чтобы сохранить простоту нашего оригинального регулярного выражения, мы отказались от проверки того, что за каждым знаком процента следуют точно две шестнадцатеричные цифры. Такого рода проверку лучше выполнить после того, как будут выделены отдельные части адреса URL. Так, мы обеспечиваем совпадение с именем хоста с помощью простого выражения `‹[a-z0-9\-._~%]+›`, которому также соответствуют адреса IPv4 (допустимые в соответствии с RFC 3986).

Кроме доменного имени и адреса IPv4 хост может определяться адресом IPv6, заключенным в квадратные скобки, или даже версией IP-адресов, возможной в будущем. Совпадение с адресами IPv6 обеспечивается с помощью выражения `‹\[[a-f0-9:.]+\]›`, а с версией, возможной в будущем, – с помощью выражения `‹\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]›`. Мы не можем проверить IP-адреса для версии, которая еще не существует, но мы могли бы более строго подойти к адресам IPv6. Однако и эту работу лучше оставить для второго регулярного выражения, которую можно выполнить после извлечения адреса из URL. Рецепт 8.17 показывает, что проверка адресов IPv6 выполняется весьма тривиально.

Номер порта, если он указан, – это простое десятичное число, отделяемое от имени хоста двоеточием. Выражение `‹:[0-9]+›` – все, что нам необходимо.

Если авторизация определена, за ней должен следовать либо абсолютный путь, либо путь вообще не указывается. Абсолютный путь начинается с символа слэша, за которым следует один или более сегментов, отделяемых друг от друга слэшами. Сегмент состоит из одной или более букв, цифр или знаков пунктуации. Здесь символы слэша не могут следовать подряд. Путь может заканчиваться символом слэша. Совпадение с такими путями обеспечивает выражение `‹(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?›`.

Если в URL отсутствует авторизация, путь может быть как абсолютным, так и относительным или отсутствовать вообще. Абсолютный путь начинается с символа слэша, тогда как относительный – нет. Поскольку на этот раз ведущий символ слэша становится необязательным, нам потребовалось написать более длинное регулярное выражение, совпадающее сразу с абсолютными и относительными путями: `‹/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?›`.

Относительные адреса URL не определяют схему адресации, а следовательно, и авторизацию. В этом случае путь становится обязательной частью адреса, при этом он может быть абсолютным или относительным. Поскольку в URL отсутствует схема, первый сегмент относительного

пути не может содержать символы двоеточия. В противном случае этот символ двоеточия интерпретировался бы как символ, отделяющий наименование схемы. Поэтому, чтобы обеспечить совпадение с путем в относительном адресе URL, нам потребовались два регулярных выражения. Совпадение с относительными путями обеспечивает выражение `‹[a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?›`. Оно очень похоже на выражение, соответствующее путям со схемой, но без авторизации. Отличия состоят в необязательности ведущего символа слэша, который отсутствует, и в содержимом первого символьного класса, из которого исключен символ двоеточия. Совпадение с абсолютным путем обеспечивает выражение `‹(/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?›`. Это то же самое регулярное выражение, которое использовалось для совпадения с путями в адресах URL, в которых задается схема адресации и авторизация, за исключением того, что звездочка, повторяющая сегменты пути, заменена на плюс. Относительные пути требуют наличия хотя бы одного сегмента.

Часть со строкой запроса является необязательной. Если она присутствует, то должна начинаться со знака вопроса. Строка запроса продолжается до первого символа решетки или до конца адреса. Поскольку знак решетки не попадает в число знаков пунктуации, допустимых в строке запроса URL, мы легко можем обеспечить сопоставление с ней с помощью выражения `‹\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*›`. Оба знака вопроса в этом регулярном выражении являются литералами. Первый из них находится за пределами символьного класса и должен экранироваться. Второй находится внутри символьного класса, где он всегда интерпретируется как литерал.

Заключительная часть URL – это фрагмент, который также является необязательным. Он начинается со знака решетки и тянется до конца адреса URL. Соответствие с ним обеспечивает выражение `‹\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*›`.

Чтобы упростить доступ к различным частям URL, мы использовали именованные сохраняющие группы. В рецепте 2.11 описывается, как работает именованное сохранение в различных диалектах регулярных выражений, рассматриваемых в этой книге. Диалекты Perl 5.10, Ruby 1.9 и .NET допускают наличие в выражении нескольких групп с одинаковыми именами. Это очень удобно в данной ситуации, потому что наше регулярное выражение предусматривает множество способов совпадения с путем в URL в зависимости от того, указывается ли схема и/или авторизация. Дав этим трем группам одно и то же имя, мы можем просто обратиться к группе с именем «path», чтобы получить путь независимо от того, указана ли в адресе URL схема и/или авторизация.

Другие диалекты не поддерживают возможность именованного сохранения, даже при том, что многие придерживаются того же синтаксиса именованных сохранений. В других диалектах все сохраняющие группы, совпадающие с путями, имеют разные имена. После того как будет

найдено совпадение, только одна из них будет действительно хранить путь. Две другие не будут участвовать в сопоставлении.

См. также

В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение необходимых частей URL.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

В рецепте 8.1 приводится простое решение, следующее более свободным правилам проверки адресов URL в сравнении со строгими правилами в RFC 3986, используемым большинством браузеров.

8.8. Извлечение схемы из адреса URL

Задача

Требуется из строки, в которой хранится URL, извлечь схему адресации. Например, необходимо извлечь текст `http` из строки `http://www.regexcookbook.com`.

Решение

Извлечение схемы из заведомо допустимого адреса URL

```
^([a-z][a-z0-9+\-.]*):
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечение схемы при проверке адреса URL

```
\A
([a-z][a-z0-9+\-.]*):
(# Авторизация и путь
 //
 ([a-z0-9\-._~%!$&'()*+,;=]+@)?                # Пользователь
 ([a-z0-9\-._~%]+                              # Имя хоста
 |\[[a-f0-9:.]+\]                              # IP-адрес версии IPv6
 |\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей версии IPv...
 (:[0-9]+)?                                    # Порт
 (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?            # Путь
```

```
|# Путь без авторизации
 (/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
)
# Строка запроса
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Фрагмент
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*):(//([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↵
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|(/?[a-z0-9\-._~%!$&'()*+,;=:@]+↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?)(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↵
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Извлечь схему из адреса URL будет проще, если известно, что испытуемый текст является допустимым адресом URL. Схема URL всегда находится в самом начале адреса URL. Соблюдение этого требования в регулярном выражении обеспечивает символ крышки (рецепт 2.5). Схема должна начинаться с буквы, за которой могут следовать дополнительные буквы, цифры, знаки плюс, дефисы и точки. Совпадение с ними мы обеспечили с помощью двух символьных классов ‹[a-z][a-z0-9+\-.]*› (рецепт 2.3).

Схема отделяется от остальной части URL символом двоеточия. Мы добавили его в регулярное выражение, чтобы гарантировать совпадение со схемой, только если адрес URL действительно начинается с нее. Относительные адреса URL не имеют схемы. Синтаксис URL, определяемый в RFC 3986, гарантирует отсутствие символов двоеточия в относительных адресах URL, если только этим двоеточиям не предшествуют символы, которые не допускается использовать в схемах. Именно по этой причине мы исключили символ двоеточия из символьного класса, используемого для сопоставления с путем, в рецепте 8.7. Если применить регулярные выражения из этого рецепта к действительному, но относительному URL, они вообще не обнаружат совпадения.

Поскольку регулярное выражение совпадает не только с названием схемы (в совпадение включается символ двоеточия), мы добавили в регулярное выражение сохраняющую группировку. Если регулярное выражение обнаружит совпадение, название схемы без двоеточия можно будет извлечь из первой (и единственной) сохраняющей группы. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9.

В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в том или ином языке программирования.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать упрощенную версию регулярного выражения из рецепта 8.7. Поскольку по условию задачи требуется извлечь схему, можно исключить из регулярного выражения сопоставление с относительными адресами URL, в которых наименование схемы не указывается. Это позволило упростить регулярное выражение.

Поскольку регулярное выражение совпадает со всем адресом URL, мы добавили дополнительную сохраняющую группировку, заключив в нее часть регулярного выражения, совпадающую со схемой. Извлекая текст, совпавший с первой сохраняющей группой, можно получить схему URL.

См. также

В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение схемы URL.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.9. Извлечение имени пользователя из URL

Задача

Требуется извлечь имя пользователя из строки, хранящей адрес URL. Например, нужно извлечь имя jan из `ftp://jan@www.regexcookbook.com`.

Решение

Извлечение имени пользователя из заведомо допустимого адреса URL

```
^[a-z0-9+\-.]+://([a-z0-9\-._~%!$&'()*+,;=]+)@
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечение имени пользователя при проверке адреса URL

```
\A
[a-z][a-z0-9+\-.]*://                          # Схема
([a-z0-9\-._~%!$&'()*+,;=]+)@                  # Пользователь
([a-z0-9\-._~%]+                               # Имя хоста
|\[[a-f0-9:.]+\]                               # IP-адрес версии IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])     # IP-адрес будущей версии IPv...
(:[0-9]+)?                                     # Порт
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?             # Путь
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?            # Строка запроса
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?            # Фрагмент
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+)@([a-z0-9\-._~%]+|↵
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↵
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Извлечь имя пользователя из адреса URL будет проще, если известно, что испытуемый текст является допустимым адресом URL. Имя пользователя, если оно присутствует в URL, располагается правее схемы и двух символов слэша, с которых в URL начинается сегмент «авторизации». Имя пользователя отделяется от следующего за ним имени хоста знаком @. Поскольку знак @ не может появляться в именах хостов, мы можем быть уверены, что извлекли имя пользователя из URL, если обнаружили знак @ после двух символов слэша и перед следующим слэшем в URL. Символы слэша не могут использоваться в именах пользователей, поэтому нам не потребовалось выполнять дополнительную проверку.

Все эти правила подразумевают, что мы очень легко можем извлечь имя пользователя, если заранее известно, что испытуемый текст представляет допустимый адрес URL. Мы пропускаем схему с помощью регулярного выражения ‹[a-z0-9+\-.]+› и ://. После этого извлекаем имя пользователя, следующее дальше. Если было найдено совпадение со знаком @, можно быть уверенными, что символы перед ним составляют имя пользователя. Символьный класс ‹[a-z0-9\-._~%!$&'()*+,;=]› перечисляет все символы, которые считаются допустимыми в именах пользователей.

Это регулярное выражение будет обнаруживать совпадение, только если URL действительно содержит имя пользователя. В этом случае в совпадение с регулярным выражением попадут обе части URL, схема и имя

пользователя. Поэтому мы добавили в регулярное выражение сохраняющую группировку. Когда регулярное выражение обнаружит совпадение, имя пользователя без символов-разделителей или других частей URL можно будет извлечь из первой (и единственной) сохраняющей группы. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать упрощенную версию регулярного выражения из рецепта 8.7. Поскольку по условию задачи требуется извлечь имя пользователя, можно исключить из регулярного выражения сопоставление с адресами URL, в которых отсутствует авторизация. Фактически регулярное выражение, которое приводится в решении, совпадает только с адресами URL, которые содержат авторизацию, включающую имя пользователя. Требование к наличию сегмента авторизации позволило упростить регулярное выражение. Более того, оно получилось даже проще, чем в рецепте 8.8.

Поскольку это регулярное выражение совпадает со всем адресом URL, мы добавили дополнительную сохраняющую группировку, заключив в нее часть регулярного выражения, совпадающую с именем пользователя. Извлекая текст, совпавший с первой сохраняющей группой, можно получить имя пользователя URL.

Если необходимо, чтобы регулярное выражение совпадало с любым допустимым адресом URL, включая те, что не содержат имени пользователя, можно воспользоваться одним из регулярных выражений в рецепте 8.7. Первое регулярное выражение в рецепте 8.7 сохраняет имя пользователя (если оно указано) в третьей сохраняющей группе. Эта группа сохраняет также символ @. Если необходимо получить имя пользователя без символа @, можно добавить в регулярное выражение еще одну сохраняющую группировку.

См. также

В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение имени пользователя.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.10. Извлечение имени хоста из URL

Задача

Требуется извлечь имя хоста из строки, хранящей адрес URL. Например, имя www.regexcookbook.com **из** http://www.regexcookbook.com.

Решение

Извлечение имени хоста из заведомо допустимого адреса URL

```
\A
[a-z][a-z0-9+\-.]*://              # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?     # Имя пользователя
([a-z0-9\-._~%]+                   # Имя хоста или IP-адрес IPv4
|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес IPv6+
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↵
\[[a-z0-9\-._~%!$&'()*+,;=:]+\])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечение имени хоста при проверке адреса URL

```
\A
[a-z][a-z0-9+\-.]*://                                # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?                       # Имя пользователя
([a-z0-9\-._~%]+                                     # Имя хоста
|\[[a-f0-9:.]+\]                                     # IP-адрес IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])          # IP-адрес будущей версии IPv...
(:[0-9]+)?                                           # Порт
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?                   # Путь
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?                  # Строка запроса
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?                  # Фрагмент
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↵
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↵
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Извлечь имя хоста из адреса URL будет проще, если известно, что испытуемый текст является допустимым адресом URL. Мы использовали якорные метасимволы ‹\A› или ‹^›, чтобы привязать совпадение к началу текста. Выражение ‹[a-z][az0-9+\-.]*://› пропускает схему, а выражение ‹([a-z0-9\-._~%!$&'()*+,;=]+@)?› пропускает необязательное имя пользователя. Сразу же вслед за ними следует имя хоста.

В RFC 3986 допускается две различные формы записи хоста. Доменные имена и адреса IPv4 указываются без квадратных скобок, тогда как адреса IPv6 и IP-адреса возможных будущих версий указываются в квадратных скобках. Эти две формы записи необходимо обрабатывать отдельно, потому что форма записи в квадратных скобках допускает большее число знаков пунктуации, чем форма записи без скобок. В частности, в квадратных скобках допускается использовать символ двоеточия, а в доменных именах и адресах IPv4 – нет. Двоеточие также используется для отделения имени хоста (в обеих формах записи) от номера порта.

Выражению ‹[a-z0-9\-._~%]+› соответствуют доменные имена и адреса IPv4. Выражение ‹\[[a-z0-9\-._~%!$&'()*+,;=:]+\]› обрабатывает адреса версии IPv6 и выше. Мы объединили эти два регулярных выражения в группу, используя конструкцию выбора (рецепт 2.8). Кроме этого, сохраняющая группа позволяет извлекать имя хоста.

Это регулярное выражение будет обнаруживать совпадение, только если URL действительно содержит имя хоста. В этом случае в совпадение с регулярным выражением попадут схема, имя пользователя и имя хоста. Когда регулярное выражение обнаружит совпадение, имя хоста без символов-разделителей или других частей URL можно будет извлечь из второй сохраняющей группы. Для адресов IPv6 в совпадение с сохраняющей группой будут включаться квадратные скобки. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать упрощенную версию регулярного выражения из рецепта 8.7. Поскольку по условию задачи требуется извлечь имя хоста, можно исключить из регулярного выражения сопоставление с адресами URL, в которых отсутствует авторизация. Это существенно упрощает регулярное выражение. Оно очень похоже на регулярное выражение, которое приводилось в рецепте 8.9. Единственное отличие в том, что теперь имя пользователя опять стало необязательным, как и в рецепте 8.7.

Кроме того, это регулярное выражение использует конструкцию выбора для различных форм записи имени хоста, заключенную в сохраняющую группу. Извлекая текст, совпавший со второй сохраняющей группой, можно получить имя хоста URL.

Если необходимо, чтобы регулярное выражение совпадало с любым допустимым адресом URL, включая те, что не содержат имени хоста, можно воспользоваться одним из регулярных выражений из рецепта 8.7. Первое регулярное выражение в рецепте 8.7 сохраняет имя хоста (если оно указано) в четвертой сохраняющей группе.

См. также

В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение адреса хоста.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.11. Извлечение номера порта из URL

Задача

Требуется извлечь номер порта из строки, хранящей адрес URL. Например, нужно извлечь число 80 из адреса `http://www.regexcookbook.com:80/`.

Решение

Извлечение номера порта из заведомо допустимого адреса URL

```
\A
[a-z][a-z0-9+\-.]*://              # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?     # Имя пользователя
([a-z0-9\-._~%]+                   # Имя хоста или адрес IPv4
|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]) # Адрес IPv6+
:(?<port>[0-9]+)                   # Номер порта
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java 7, PCRE 7, Perl 5.10, Python, Ruby 1.9

```
\A
[a-z][a-z0-9+\-.]*://              # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?     # Имя пользователя
([a-z0-9\-._~%]+                   # Имя хоста или адрес IPv4
|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]) # Адрес IPv6+
:(?P<port>[0-9]+)                  # Номер порта
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE, Perl 5.10, Python

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?↵
([a-z0-9\-._~%]+|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]):([0-9]+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечение номера порта при проверке адреса URL

```
\A
[a-z][a-z0-9+\-.]*://                              # Схема
([a-z0-9\-._~%!$&'()*+,;=]+@)?                     # Имя пользователя
([a-z0-9\-._~%]+                                   # Имя хоста
|\[[a-f0-9:.]+\]                                   # Адрес IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # IP-адрес будущей версии IPv...
:([0-9]+)                                          # Порт
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?                 # Путь
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?                # Строка запроса
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?                # Фрагмент
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*:\/\/([a-z0-9\-._~%!$&'()*+,;=]+@)?↵
([a-z0-9\-._~%]+|\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]↵
+\]):([0-9]+)(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?↵
(\?[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?(#[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Извлечь номер порта из адреса URL будет проще, если известно, что испытуемый текст является допустимым адресом URL. Мы использовали якорные метасимволы ‹\A› или ‹^›, чтобы привязать совпадение к началу текста. Выражение ‹[a-z][a-z0-9+\-.]*://› пропускает схему, а выражение ‹([a-z0-9\-._~%!$&'()*+,;=]+@)?› пропускает необязательное имя пользователя. Выражение ‹([a-z0-9\-._~%]+|\[[a-z0-9\-._~%!$&'()*+,;=:]+\])› пропускает имя хоста.

Номер порта отделяется от имени хоста символом двоеточия, который мы добавили в регулярное выражение как литерал. Номер порта – это просто строка цифр, совпадение с которой легко обеспечить с помощью выражения ‹[0-9]+›.

Это регулярное выражение будет обнаруживать совпадение, только если URL действительно содержит номер порта. В этом случае в совпаде-

ние с регулярным выражением попадут схема, имя пользователя, имя хоста и номер порта. Когда регулярное выражение обнаружит совпадение, номер порта без символов-разделителей или других частей URL можно будет извлечь из третьей сохраняющей группы.

Другие две группы обеспечивают необязательность имени пользователя и объединение альтернативных форм записи имени хоста. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Если заранее неизвестно, что испытуемый текст является допустимым адресом URL, можно использовать упрощенную версию регулярного выражения из рецепта 8.7. Поскольку по условию задачи требуется извлечь номер порта, можно исключить из регулярного выражения сопоставление с адресами URL, в которых этот номер отсутствует. Это существенно упрощает регулярное выражение. Оно очень похоже на регулярное выражение, которое приводилось в рецепте 8.10.

Единственное отличие состоит в том, что теперь номер порта уже не является необязательным, а еще мы переместили границу сохраняющей группировки, соответствующей номеру порта, исключив из нее двоеточие, отделяющее номер порта от имени хоста. Номер сохраняющей группы – 3.

Если необходимо, чтобы регулярное выражение совпадало с любым допустимым адресом URL, включая те, что не содержат номера порта, можно воспользоваться одним из регулярных выражений из рецепта 8.7. Первое регулярное выражение в рецепте 8.7 сохраняет номер порта (если он указан) в пятой сохраняющей группе.

См. также

В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение номера порта.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.12. Извлечение пути из адреса URL

Задача

Требуется извлечь путь из строки, хранящей адрес URL. Например, нужно извлечь фрагмент /index.html **из адреса** http://www.regexcookbook.com/index.html **или из адреса** /index.html#fragment**.**

Решение

Извлечь путь из адреса URL, если известно, что строка хранит допустимый адрес URL. Следующее регулярное выражение совпадает со всеми URL, даже с теми, в которых путь не указан:

```
\A
# Пропустить схему и авторизацию, если имеются
([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?
# Путь
([a-z0-9\-._~%!$&'()*+,;=:@/]*)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?([a-z0-9\-._~%!$&'()*+,;=:@/]*)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечь путь из адреса URL, если известно, что строка хранит допустимый адрес URL. Совпадение должно обеспечиваться только с адресами URL, содержащими путь:

```
\A
# Пропустить схему и авторизацию, если имеются
([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?
# Путь
(/?[a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|/)
# Строка запроса, фрагмент или конец URL
([#?]|\Z)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?(/?[a-z0-9\-._~%!$&'()*+,;=@]+↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|/)([#?]|$)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Извлечь путь из адреса URL, если известно, что строка хранит допустимый адрес URL. Чтобы обеспечить совпадение только с адресами URL, содержащими путь, использовать атомарную группировку:

```
\A
# Пропустить схему и авторизацию, если имеются
(?>([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?)
# Путь
([a-z0-9\-._~%!$&'()*+,;=:@/]+)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Ruby

Обсуждение

Регулярное выражение, извлекающее путь, можно существенно упростить, если известно, что испытуемый текст является допустимым адресом URL. В рецепте 8.7 предусмотрено три различных способа совпадения с путем в зависимости от наличия сегментов схемы и/или авторизации в URL, но в регулярном выражении, извлекающем путь из заведомо допустимого адреса URL, достаточно одного способа сопоставления с путем.

Регулярное выражение начинается с якорного метасимвола ‹\A› или ‹^›, чтобы привязать совпадение к началу текста. Выражение ‹[a-z][a-z0-9+\-.]*:› перешагивает через схему, а выражение ‹//[^/?#]+› — через авторизацию. Мы смогли использовать такое простое выражение, соответствующее авторизации, потому что уже знаем, что URL допустим, и нам не требуется извлекать имя пользователя, имя хоста или номер порта из авторизации. Сегмент авторизации начинается с двух символов слэша и простирается до начала пути (символ слэша), строки запроса (знак вопроса) или фрагмента (символ решетки). Инвертированный символьный класс обеспечивает совпадение с любыми символами, кроме первого символа слэша, знака вопроса и решетки (рецепт 2.3).

Поскольку сегмент авторизации является необязательным, мы поместили его в группу, за которой следует квантификатор «знак вопроса»: ‹(//[^/?#]+)?›. Схема также является необязательным элементом URL. Если схема опущена, авторизация тоже должна быть опущена. Чтобы обеспечить соблюдение этого правила, мы поместили части регулярного выражения, совпадающие со схемой и необязательным сегментом авторизации, в другую группу, которую также сделали необязательной с помощью вопросительного знака.

Нам заранее известно, что испытуемый текст является допустимым адресом URL, поэтому совпадение с сегментом пути легко можно обеспечить с помощью единственного символьного класса ‹[a-z0-9\-._~%!$&'()*+,;=:@/]*›, который включает символ слэша. Нам не требуется проверять наличие двойных символов слэша, не допустимых в пути.

Мы целенаправленно использовали звездочку, а не плюс в качестве квантификатора символьного класса для сегмента пути. Может показаться странным, что совпадение с путем сделано необязательным в регулярном выражении, единственная цель которого извлечь путь из

URL. Фактически совпадение с сегментом пути должно быть необязательным из-за упрощений, сделанных нами для перескакивания через сегменты схемы и авторизации.

В универсальном регулярном выражении в рецепте 8.7 имеется три разных способа сопоставления с сегментом пути в зависимости от наличия в URL схемы и/или авторизации. Это гарантирует, что схема по ошибке не будет воспринята как путь.

Но сейчас мы стараемся упростить выражение, используя единственный символьный класс для сегмента пути. Рассмотрим URL `http://www.regexcookbook.com`, где имеются сегменты схемы и авторизации, но отсутствует путь. Первая часть нашего регулярного выражения благополучно совпадет со схемой и авторизацией. Механизм регулярных выражений попытается сопоставить символьный класс для пути, но в тексте больше не осталось символов. Если путь объявлен необязательным (с помощью квантификатора «звездочка»), механизм регулярных выражений будет вполне удовлетворен отсутствием символов для пути. Он достигнет конца регулярного выражения и объявит, что общее совпадение было найдено.

Но если бы совпадение с символьным классом пути было обязательным, механизм регулярных выражений начал бы выполнять возвраты. (Те, кто не знаком с механизмом возвратов, могут обратиться к рецепту 2.13.) Он вспомнит, что части регулярного выражения, совпадающие с сегментами схемы и авторизации, объявлены необязательными, поэтому механизм скажет: попробуем еще раз, но на этот раз исключим возможность совпадения с `‹(//[^/?#]+)?›`. Тогда подвыражение `‹[a-z0-9\-._~%!$&'()*+,;=:@/]+›` совпадет с фрагментом `//www.regexcookbook.com`, интерпретировав его как путь, что совершенно не то, что нам требуется. Если бы для сопоставления с сегментом пути использовалось более точное регулярное выражение, не допускающее появления двойных символов слэша, механизм регулярных выражений смог бы выполнить возврат еще раз и попробовать вариант с отсутствующей схемой. Тогда, при использовании точного выражения для пути, фрагмент `http` адреса был бы воспринят как путь. Для предотвращения этого нам пришлось бы добавить дополнительную проверку наличия строки запроса и фрагмента за сегментом пути. Если реализовать все это, мы получим регулярные выражения, обозначенные в разделе «Решение» как «совпадающие только с адресами URL, действительно содержащими путь». Они получились немного сложнее, только чтобы обеспечить отсутствие совпадения с адресом URL без пути.

Если используемый диалект регулярных выражений поддерживает атомарную группировку, с ее помощью можно получить более простое решение. Атомарную группировку (рецепт 2.14) поддерживают все диалекты, рассматриваемые в книге, кроме JavaScript и Python. Атомарная группировка сообщает механизму регулярных выражений, что не следует выполнять возврат. Если подвыражения, совпадающие с сег-

ментами схемы и авторизации, заключить в атомарную группу, механизм регулярных выражений будет вынужден сохранить совпадения со схемой и авторизацией, как только они будут обнаружены, даже если в результате этого не останется символов для символьного класса, обеспечивающего совпадение с путем. Это решение такое же эффективное, как и предыдущее – с необязательным сегментом пути.

Путь можно будет извлечь из третьей сохраняющей группы при использовании любого регулярного выражения из этого рецепта. Если использовать первые два выражения, в которых совпадение с путем является необязательным, в третьей сохраняющей группе может возвращаться пустая строка или значение `null` в JavaScript.

Если заранее неизвестно, что испытуемый текст является допустимым адресом URL, можно воспользоваться регулярным выражением из рецепта 8.7. При работе с платформой .NET можно применить выражение, работающее только с диалектом .NET и содержащее три группы с именем «path» в трех подвыражениях, способных совпасть с сегментом пути в URL. В других диалектах, поддерживающих именованные сохранения, путь будет сохраняться в одной из трех групп: «hostpath», «schemepath» или «relpath». Так как только одна из трех групп действительно будет содержать что-нибудь, для извлечения пути можно воспользоваться простой хитростью, выполнив операцию конкатенации строк, возвращаемых тремя группами. Две из них будут возвращать пустые строки, поэтому фактически никакой конкатенации выполняться не будет.

Если используемый диалект не поддерживает именованное сохранение, можно использовать первое регулярное выражение из рецепта 8.7. Оно сохраняет путь в группе 6, 7 или 8. В этом случае также можно использовать трюк с конкатенацией фрагментов, сохраняемых этими тремя группами, потому что две из них будут возвращать пустые строки. Однако в JavaScript этот прием работать не будет. Для групп, не участвовавших в сопоставлении, в JavaScript возвращается значение `undefined`.

В рецепте 3.9 приводится немало информации об извлечении текста, совпавшего с нумерованными или именованными сохраняющими группами в разных языках программирования.

См. также

Рецепт 3.9, в котором описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение пути.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные

группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.13. Извлечение строки запроса из URL

Задача

Требуется извлечь строку запроса из строки, хранящей адрес URL. Например, нужно извлечь фрагмент param=value **из** http://www.regexcookbook.com?param=value **или из** /index.html?param=value**.**

Решение

```
^[^?#]+\?([^#]+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение строки запроса из URL – достаточно тривиальная задача, если заведомо известно, что испытуемый текст является допустимым адресом URL. Запрос отделяется от предшествующей части URL знаком вопроса, то есть первым знаком вопроса, встретившимся в URL. Благодаря этому с помощью регулярного выражения ‹^[^?#]+\?› можно сразу же перейти к первому вопросительному знаку. Знак вопроса в регулярных выражениях интерпретируется как метасимвол только за пределами символьных классов, но не внутри, поэтому нам потребовалось экранировать литерал знака вопроса за пределами символьного класса. Первый символ ‹^› – это якорь (рецепт 2.5), тогда как второй символ ‹^› инвертирует символьный класс (рецепт 2.3).

Знак вопроса может появляться в адресах URL как часть (необязательного) фрагмента, следующего за строкой запроса. Поэтому, чтобы убедиться, что это первый знак вопроса в URL и он не является частью фрагмента в адресе URL, в котором отсутствует строка запроса, вместо простого подвыражения ‹\?› пришлось использовать ‹^[^?#]+\?›.

Строка запроса простирается до начала фрагмента или до конца строки URL, если в ней отсутствует сегмент фрагмента. Фрагмент отделяется от остальной части URL знаком решетки. Так как знак решетки не допускается нигде, кроме фрагмента, выражения ‹[^#]+› будет вполне достаточно для совпадения с запросом. Инвертированный символьный класс совпадает с любыми символами до первого встретившегося знака решетки или до конца испытуемого текста, если в нем нет ни одного знака решетки.

Это регулярное выражение будет совпадать только с адресами URL, которые действительно содержат строку запроса. Когда будет обнаружи-

ваться совпадение с URL, в его состав будет входить все от начала URL, поэтому мы поместили часть ‹[^#]+› регулярного выражения, совпадающую со строкой запроса, в сохраняющую группу. Когда регулярное выражение обнаружит совпадение, строку запроса без символов-разделителей или других частей URL можно будет извлечь из первой (и единственной) сохраняющей группы. Подробнее о сохраняющей группировке рассказывается в рецепте 2.9. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать одно из регулярных выражений из рецепта 8.7. Первое регулярное выражение в рецепте 8.7 сохраняет строку запроса, если она присутствует в строке URL, в сохраняющей группе под номером 12.

См. также

Рецепт 3.9, в котором описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение строки запроса.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.14. Извлечение фрагмента из URL

Задача

Требуется извлечь сегмент определенного фрагмента из строки, хранящей адрес URL. Например, нужно извлечь фрагмент `top` **из** `http://www.regexcookbook.com#top` **или из** `/index.html#top`.

Решение

```
#(.+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение такого фрагмента из URL – достаточно тривиальная задача, если заведомо известно, что испытуемый текст является допустимым

адресом URL. Фрагмент отделяется от предшествующей части URL знаком решетки. Фрагмент – это единственная часть URL, где допускается присутствие знака решетки, и это всегда последняя часть URL. Поэтому мы легко можем извлечь фрагмент, отыскав первый знак решетки и выбрав все символы до конца строки. С этой задачей прекрасно справляется выражение ‹#.+›. Не забудьте выключить режим свободного форматирования, в противном случае потребуется экранировать литерал знака решетки символом обратного слэша.

Это регулярное выражение будет совпадать только с адресами URL, которые действительно содержат фрагмент. В совпадении будет присутствовать только фрагмент, но оно включает в себя знак решетки, отделяющий фрагмент от остальной части URL. Чтобы получить возможность извлекать только сам фрагмент без отделяющего символа #, в решение была добавлена сохраняющая группа.

Если заранее неизвестно, является ли испытуемый текст допустимым адресом URL, можно использовать одно из регулярных выражений из рецепта 8.7. Первое регулярное выражение в рецепте 8.7 сохраняет фрагмент, если он присутствует в строке URL, в сохраняющей группе под номером 13.

См. также

Рецепт 3.9, в котором описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение фрагмента.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.15. Проверка доменных имен

Задача

Необходимо проверить, выглядит ли испытуемый текст как полное квалифицированное доменное имя, или отыскать такие доменные имена в текстовом документе.

Решение

Проверить, выглядит ли строка как допустимое доменное имя:

```
^([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Отыскать допустимые доменные имена в текстовом документе:

```
\b([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверить, не превышает ли длина каждой части доменного имени 63 символов:

```
\b((?=[a-z0-9-]{1,63}\.)[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,63}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Разрешить использование интернационализированных доменных имен в кодировке punycode:

```
\b((xn--)?[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Проверить, не превышает ли длина каждой части доменного имени 63 символов, и разрешить использование интернационализированных доменных имен в кодировке punycode:

```
\b((?=[a-z0-9-]{1,63}\.)(xn--)?[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,63}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Доменное имя имеет вид `domain.tld`, или `subdomain.domain.tld`, или может включать любое число дополнительных поддоменов. Домен верхнего уровня (top-level domain, `tld`) состоит из двух или более букв. Это самая простая часть регулярного выражения: `‹[a-z]{2,}›`.

Имена доменов и любых поддоменов состоят из букв, цифр и дефисов. Дефисы не могут следовать парами и не могут быть первыми символами в именах доменов. Совпадение с ними обеспечивается регулярным выражением `‹[a-z0-9]+(-[a-z0-9]+)*›`. Это регулярное выражение допускает любое число букв и цифр, за которыми может следовать любое число групп символов, состоящих из дефисов, за которыми следуют другие последовательности букв и цифр. Необходимо помнить, что внутри символьных классов дефис интерпретируется как метасимвол (рецепт 2.3), но за их пределами он воспринимается как обычный сим-

вол, поэтому нам не потребовалось экранировать дефисы в этом регулярном выражении.

Имена доменов и поддоменов отделяются друг от друга символом точки, совпадение с которой обеспечивает фрагмент ‹\.› в регулярном выражении. Так как в URL помимо имени домена может присутствовать любое количество имен поддоменов, мы поместили подвыражение, совпадающее с именем домена и с литералом точки в повторяемую группу: ‹([a-z0-9]+(-[a-z0-9]+)*\.)+›. Поскольку имена поддоменов следуют тому же синтаксису, что и имена доменов, эта группа обрабатывает и те, и другие имена.

Если необходимо проверить, является ли заданная строка допустимым именем домена, достаточно добавить якорные метасимволы в начало и в конец регулярного выражения, чтобы обеспечить его привязку к началу и концу текста. Сделать это можно с помощью ‹^› и ‹$› во всех диалектах, за исключением Ruby, или с помощью ‹\A› и ‹\Z› во всех диалектах, за исключением JavaScript. Подробнее об этих якорных метасимволах рассказывается в рецепте 2.5.

Если потребуется отыскать доменные имена в текстовом документе, можно добавить границы слова ‹\b› (рецепт 2.6).

Первый набор регулярных выражений не проверяет, превышает ли длина каждой части доменного имени 63 символа. Реализовать такую проверку не так просто, потому что наше регулярное выражение, совпадающее с каждой частью доменного имени, ‹[a-z0-9]+(-[a-z0-9]+)*›, содержит в себе три квантификатора и нет никакой возможности сообщить механизму регулярных выражений, что в сумме они не должны производить более 63 повторений.

Можно было бы использовать выражение ‹[-a-z0-9]{1,63}›, чтобы ограничить длину каждой части доменного имени 63 символами, или ‹\b([-a-z0-9]{1,63}\.)+[a-z]{2,63}› — для всего доменного имени. Но тогда мы лишились бы возможности исключать доменные имена с символами дефиса в недопустимых местах.

Однако можно задействовать опережающую проверку, чтобы дважды проверить совпадение с одним и тем же текстом. Те, кто не знаком с опережающей проверкой, могут сначала ознакомиться с рецептом 2.16. Мы использовали то же самое регулярное выражение ‹[a-z0-9]+(-[a-z0-9]+)*\.›, совпадающее с доменным именем, в котором дефисы находятся в допустимых местах, и добавили в опережающую проверку выражение ‹[-a-z0-9]{1,63}\.›, чтобы убедиться, что длина имени не превышает 63 символа. В результате получилось выражение ‹(?=[-a-z0-9]{1,63}\.)[a-z0-9]+(-[a-z0-9]+)*\.›.

Опережающая проверка ‹(?=[-a-z0-9]{1,63}\.)› сначала убеждается, что до ближайшего символа точки имя содержит от 1 до 63 букв, цифр и дефисов. Точка в опережающей проверке имеет большое значение. Без нее доменные имена длиннее 63 символов по-прежнему могли бы удовлетво-

рять ограничению в 63 символа, накладываемому опережающей проверкой. Только добавив литерал точки внутрь опережающей проверки, мы можем обеспечить соблюдение ограничения длины 63 символами.

Опережающая проверка не поглощает символы совпавшего с ней текста. Поэтому если опережающая проверка преуспевает, выражение `‹[a-z0-9]+(-[a-z0-9]+)*\.›` применяется к тексту, уже совпавшему с опережающей проверкой. Мы уже убедились, что эта часть имени не превышает 63 символов в длину, и теперь осталось лишь убедиться, что она представляет собой допустимую комбинацию дефисов и других символов.

Интернационализированные доменные имена (Internationalized Domain Names, IDN) теоретически могут содержать почти любые символы. Фактически же список допустимых символов зависит от регистратора, управляющего доменом верхнего уровня. Например, в домене `.es` допускается использовать доменные имена, содержащие испанские символы.

На практике интернационализированные доменные имена часто кодируются с помощью схемы, называемой *punycode*. Несмотря на всю сложность алгоритма punycode, здесь важно упомянуть, что в результате его применения получаются доменные имена, состоящие из букв, цифр и дефисов и отвечающие требованиям, которые мы уже удовлетворили в нашем регулярном выражении для доменных имен. Единственное отличие в том, что доменные имена, воспроизводимые алгоритмом punycode, начинаются с префикса `xn--`. Чтобы добавить поддержку таких доменов в наше регулярное выражение, нам достаточно добавить `‹(xn--)?›` в группу, совпадающую с частями доменного имени.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

8.16. Сопоставление с адресами IPv4

Задача

Необходимо проверить, является ли некоторая строка допустимым адресом IPv4 в форме записи 255.255.255.255. Дополнительно необходимо преобразовать этот адрес в 32-битное целое число.

Решение

Регулярное выражение

Простое регулярное выражение, выполняющее проверку IP-адресов:

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, выполняющее проверку IP-адресов и допускающее наличие ведущих нулей:

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, выполняющее проверку IP-адресов и не допускающее наличие ведущих нулей:

```
^(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}↵
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Простое регулярное выражение, извлекающее IP-адреса из текстового документа:

```
\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, извлекающее IP-адреса из текстового документа и допускающее наличие ведущих нулей:

```
\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, извлекающее IP-адреса из текстового документа и не допускающее наличие ведущих нулей:

```
\b(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}↵
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Простое регулярное выражение, сохраняющее четыре части IP-адреса:

```
^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, сохраняющее четыре части IP-адреса и допускающее наличие ведущих нулей:

```
^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↵
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↵
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↵
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Точное регулярное выражение, сохраняющее четыре части IP-адреса и не допускающее наличие ведущих нулей:

```
^(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.↵
(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.↵
(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.↵
(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])$
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Perl

```
if ($subject =~ m/^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})/)
{
    $ip = $1 << 24 | $2 << 16 | $3 << 8 | $4;
}
```

Обсуждение

IP-адреса в версии 4 обычно записываются в формате 255.255.255.255, где каждая часть должна быть представлена числом от 0 до 255. Сопоставление с такими IP-адресами реализуется очень просто.

В решении представлены четыре регулярных выражения. Два из них обозначены как «простые», а два других – как «точные».

В простых регулярных выражениях для сопоставления с каждым из четырех блоков цифр IP-адреса используется подвыражение ‹[0-9]{1,3}›. Оно фактически совпадает с числами в диапазоне от 0 до 999, а не от 0 до 255. Простые регулярные выражения более эффективны, когда заранее известно, что строка содержит только допустимые IP-адреса, и требуется только отделить IP-адреса от остального текста.

В точных регулярных выражениях для сопоставления с каждым из четырех блоков цифр IP-адреса используется подвыражение ‹25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?›. Это регулярное выражение точно соответствует числу в диапазоне от 0 до 255, содержащему необязательный ведущий ноль для чисел в диапазоне от 10 до 99 и два необязательных ведущих нуля для чисел в диапазоне от 0 до 9. Выражение ‹25[0-5]› совпадает

с числами от 250 до 255, выражение ‹2[0-4][0-9]› совпадает с числами от 200 до 249, а выражение ‹[01]?[0-9][0-9]?› совпадает с числами от 0 до 199, включая необязательные ведущие нули. Подробнее о сопоставлении с числовыми диапазонами рассказывается в рецепте 6.7.

Хотя многие приложения принимают IP-адреса с ведущими нулями, строго говоря, ведущие нули недопустимы в адресах IPv4. Поэтому точные версии были улучшены, и в их основу положено выражение ‹25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9]›, совпадающее с числами от 0 до 255 без ведущих нулей. Сопоставление с числами от 200 до 255 не изменилось и выполняется точно так же. Теперь вместо выражения ‹[01]?[0-9][0-9]?› для сопоставления с числами от 0 до 99 используется выражение ‹1[0-9][0-9]|[1-9]?[0-9]› с двумя альтернативами. Альтернатива ‹1[0-9][0-9]› соответствует диапазону от 100 до 199, а альтернатива ‹[1-9]?[0-9]› соответствует диапазону от 0 до 99. Объявив первую цифру необязательной, нам удалось одной альтернативой охватить однозначные и двузначные числа.

Если необходимо проверить, является ли вся строка допустимым IP-адресом, следует использовать одно из регулярных выражений, начинающихся символом крышки и заканчивающихся знаком доллара. Эти якорные метасимволы, совпадающие с началом и концом строки, описываются в рецепте 2.5. Если необходимо выполнить поиск IP-адресов в текстовом документе, следует использовать регулярные выражения, начинающиеся и заканчивающиеся метасимволом границы слова ‹\b› (рецепт 2.6).

Первые четыре регулярных выражения имеют вид ‹(?:number\.){3}number›. Первые три числа в IP-адресе совпадают с несохраняющей группой (рецепт 2.9), которая повторяется три раза (рецепт 2.12). Группе соответствует число и литерал точки, которая трижды встречается в IP-адресе. Последней части регулярного выражения соответствует заключительное число IP-адреса. Использование несохраняющей группы и трехкратное ее повторение делает регулярное выражение короче и эффективнее.

Чтобы преобразовать текстовое представление IP-адреса в целое число, необходимо сохранить все четыре числа отдельно. Это реализовано в двух последних регулярных выражениях. Вместо одной группы, повторяющейся трижды, в них используются четыре сохраняющих группы, по одной для каждого числа. Это единственный способ сохранить по отдельности все четыре числа, присутствующие в IP-адресе.

После сохранения чисел объединить их в 32-битное целое число не составляет труда. Текст, совпавший с четырьмя сохраняющими группами регулярного выражения, в языке Perl хранится в специальных переменных $1, $2, $3 и $4. Порядок извлечения содержимого сохраняющих групп в других языках программирования описывается в рецепте 3.9. В языке Perl строковые переменные, соответствующие сохраняющим группам, автоматически преобразуются в числа при применении к ним оператора битового сдвига (<<). В других языках программирования

может потребоваться вызвать метод `String.toInteger()` или похожий на него, прежде чем можно будет выполнить сдвиг чисел и объединить их.

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

8.17. Сопоставление с адресами IPv6

Задача

Необходимо проверить, является ли некоторая строка допустимым адресом IPv6 в стандартной, компактной и/или смешанной форме записи.

Решение

Стандартная форма записи

Обеспечить совпадение с адресом IPv6 в стандартной форме записи, когда адрес состоит из восьми 16-битовых слов в шестнадцатеричной нотации, разделенных двоеточиями (например, `1762:0:0:0:0:B03:1:AF18`). Ведущие нули являются необязательными.

Проверить, является ли весь испытуемый текст адресом IPv6 в стандартной форме записи:

```
^(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}\Z
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Отыскать в текстовом документе адреса IPv6 в стандартной форме записи:

```
(?<![:.\w])(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку. Нам пришлось удалить проверку в начале регулярного выраже-

ния, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Частично проверка выполняется с помощью границы слова:

```
\b(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Смешанная форма записи

Обеспечить совпадение с адресом IPv6 в смешанной форме записи, когда адрес состоит из шести 16-битных слов в шестнадцатеричной нотации, за которыми следуют четыре байта в десятичной нотации. Слова отделяются двоеточиями, а байты – точками. Слова и байты отделяются двоеточием. Ведущие нули в шестнадцатеричных словах и в десятичных байтах являются необязательными. Данная форма записи используется в случае смешивания адресов формата IPv4 и IPv6 и когда адреса IPv4 дополняются до формата IPv6. Пример адреса IPv6 в смешанной форме записи: 1762:0:0:0:0:B03:127.32.67.15.

Проверить, является ли весь испытуемый текст адресом IPv6 в смешанной форме записи:

```
^(?:[A-F0-9]{1,4}:){6}(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])↵
\.){3}(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Отыскать в текстовом документе адреса IPv6 в смешанной форме записи:

```
(?<![:.\w])(?:[A-F0-9]{1,4}:){6}↵
(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}↵
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку. Нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Частично проверка выполняется с помощью границы слова:

```
\b(?:[A-F0-9]{1,4}:){6}(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])↵
\.){3}(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Стандартная или смешанная форма записи

Обеспечить совпадение с адресом IPv6 в стандартной или смешанной форме записи.

Проверить, является ли весь испытуемый текст адресом IPv6 в стандартной или смешанной форме записи:

```
\A                                                          # Начало текста
(?:[A-F0-9]{1,4}:){6}                                          # 6 слов
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}                                 # 2 слова
|  (?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3} # или 4 байта
   (?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
)\Z                                                         # Конец текста
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:[A-F0-9]{1,4}:){6}(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}|↵
(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}↵
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9]))$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Отыскать в текстовом документе адреса IPv6 в стандартной или смешанной форме записи:

```
(?<![:.\w])                                                 # Привязка к адресу
(?:[A-F0-9]{1,4}:){6}                                          # 6 слов
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}                                 # 2 слова
|  (?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3} # или 4 байта
   (?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
)(?![:.\w])                                                 # Привязка к адресу
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку. Нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Частично проверка выполняется с помощью границы слова:

```
\b                                                          # Граница слова
(?:[A-F0-9]{1,4}:){6}                                          # 6 слов
   (?:[A-F0-9]{1,4}:[A-F0-9]{1,4}                              # 2 слова
    | (?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3} # или 4 байта
     (?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
)\b                                                         # Граница слова
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
\b(?:[A-F0-9]{1,4}:){6}(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}|↵
(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}↵
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9]))\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Компактная форма записи

Обеспечить совпадение с адресом IPv6 в компактной форме записи. Компактная форма записи идентична стандартной, за исключением того, что в компактной форме одна из последовательностей из одного или более слов, содержащих ноль, может быть опущена, но при этом двоеточия до и после опущенных нулей остаются. Опознать адрес в компактной форме записи можно по двум двоеточиям, следующим подряд. Опущена может быть только одна последовательность нулей, в противном случае будет невозможно определить, как много слов было опущено в каждой последовательности. Если опущенная последовательность нулей находится в начале или в конце IP-адреса, он будет начинаться или оканчиваться двумя двоеточиями. Если в IP-адресе все числа равны нулю, такой адрес в компактной форме будет содержать всего два двоеточия без каких либо цифр.

Например, 1762::B03:1:AF18 — это адрес 1762:0:0:0:0:B03:1:AF18 в компактной форме записи. Регулярные выражения в этом разделе будут совпадать как с компактной, так и со стандартной формой записи адреса IPv6. Проверить, является ли весь испытуемый текст адресом IPv6 в стандартной или компактной форме записи:

```
\A(?:
 # Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Компактная, не более 7 двоеточий
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    \Z) # и конец текста
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
 # Компактная с 8 двоеточиями
|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7}
)\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}:){0,7}↵
[A-F0-9]{0,4}$)(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)↵
|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7})$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Отыскать в текстовом документе адреса IPv6 в стандартной или компактной форме записи:

```
(?<![:.\w])(?:
 # Стандартная
```

```
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Компактная, не более 7 двоеточий
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
 # Компактная с 8 двоеточиями
|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7}
)(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку, поэтому нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Мы не можем использовать границу слова, потому что адрес может начинаться с символа двоеточия, который не является символом слова:

```
(?:
 # Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Компактная, не более 7 двоеточий
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
 # Компактная с 8 двоеточиями
|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7}
)(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}:){0,7}↵
[A-F0-9]{0,4}(?![:.\w]))(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)↵
|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7})(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Смешанная компактная форма записи

Обеспечить совпадение с адресом IPv6 в смешанной компактной форме записи. Компактная смешанная форма записи идентична смешанной, за исключением того, что в компактной форме записи одна из последовательностей из одного или более слов, содержащих ноль, может быть опущена, при этом двоеточия до и после опущенных нулей оставлены. Четыре десятичных байта должны присутствовать всегда, даже если они равны нулю. Опознать адрес в компактной смешанной форме записи

можно по двум двоеточиям, следующим подряд в первой части адреса, и по трем точкам во второй части. Опущена может быть только одна последовательность нулей, в противном случае будет невозможно определить, как много слов было опущено в каждой последовательности. Если опущенная последовательность нулей находится в начале IP-адреса, он будет начинаться двумя двоеточиями, а не цифрой.

Например, 1762::B03:127.32.67.15 – это компактная смешанная форма записи адреса 1762:0:0:0:0:B03:127.32.67.15. Регулярные выражения в этом разделе будут совпадать с адресами IPv6 в смешанной форме записи, как в компактной, так и в некомпактной.

Проверить, является ли весь испытуемый текст адресом IPv6 в компактной или некомпактной смешанной форме записи:

```
\A
(?:
 # Некомпактная
 (?:[A-F0-9]{1,4}:){6}
 # Компактная, число двоеточий не более 6
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
    \Z)                           # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 # Компактная с 7 двоеточиями и 5 числами
|::(?:[A-F0-9]{1,4}:){5}
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python

```
^(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.)↵
{3}[0-9]{1,3}$)(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)↵
|::(?:[A-F0-9]{1,4}:){5})(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|↵
[1-9]?[0-9])\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Отыскать в текстовом документе адреса IPv6 в смешанной компактной или некомпактной форме записи:

```
(?<![:.\w])
(?:
 # Некомпактная
 (?:[A-F0-9]{1,4}:){6}
 # Компактная, число двоеточий не более 6
```

```
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
    (?![:.\w]))                   # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 # Компактная с 7 двоеточиями и 5 числами
|::(?:[A-F0-9]{1,4}:){5}
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку, поэтому нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Мы не можем использовать границу слова, потому что адрес может начинаться с символа двоеточия, который не является символом слова:

```
(?:
 # Некомпактная
 (?:[A-F0-9]{1,4}:){6}
 # Компактная, не более 6 двоеточий
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
    (?![:.\w]))                   # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 # Компактная с 7 двоеточиями и 5 числами
|::(?:[A-F0-9]{1,4}:){5}
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3}↵
[0-9]{1,3}(?![:.\w]))(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)↵
|::(?:[A-F0-9]{1,4}:){5})(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?↵
[0-9])\.){3}(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Стандартная, смешанная или компактная форма записи

Обеспечить совпадение с адресом IPv6 в любой из форм записи, описанных выше: стандартной, смешанной и компактной.

Проверить, является ли весь испытуемый текст адресом IPv6:

```
\A(?:
# Смешанная
 (?:
  # Некомпактная
  (?:[A-F0-9]{1,4}:){6}
  # Компактная, не более 6 двоеточий
 |(?=(?:[A-F0-9]{0,4}:){0,6}
     (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
     \Z)                           # и привязка
  # и не более 1 пары двоеточий
  (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
  # Компактная с 7 двоеточиями и 5 числами
 |::(?:[A-F0-9]{1,4}:){5}
 )
 # 255.255.255.
 (?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}
 # 255
 (?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
|# Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Компактная, не более 7 двоеточий
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    \Z) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
 # Компактная с 8 двоеточиями
|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7}
)\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}↵
\.){3}[0-9]{1,3}$)((0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)↵
|::(?:[A-F0-9]{1,4}:){5})(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|↵
[1-9]?[0-9])\.){3}(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])|↵
(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}:){0,7}↵
[A-F0-9]{0,4}$)((0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)|↵
(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7})$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Отыскать в текстовом документе адреса IPv6 в стандартной, смешанной или компактной форме записи:

```
(?<![:.\w])(?:
# Смешанная
 (?:
  # Некомпактная
  (?:[A-F0-9]{1,4}:){6}
  # Компактная, не более 6 двоеточий
 |(?=(?:[A-F0-9]{0,4}:){0,6}
     (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
     (?![:.\w]))                   # и привязка
  # и не более 1 пары двоеточий
  (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
  # Компактная с 7 двоеточиями и 5 числами
 |::(?:[A-F0-9]{1,4}:){5}
 )
 # 255.255.255.
 (?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}
 # 255
 (?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
|# Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Компактная, не более 7 двоеточий
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
 # Компактная с 8 двоеточиями
|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7}
)(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Диалекты JavaScript и Ruby 1.8 не поддерживают ретроспективную проверку, поэтому нам пришлось удалить проверку в начале регулярного выражения, которая препятствует совпадению с адресами IPv6 внутри длинных последовательностей шестнадцатеричных цифр и двоеточий. Мы не можем использовать границу слова, потому что адрес может начинаться с символа двоеточия, который не является символом слова:

```
(?:
 # Смешанная
 (?:
  # Некомпактная
  (?:[A-F0-9]{1,4}:){6}
  # Компактная, не более 6 двоеточий
 |(?=(?:[A-F0-9]{0,4}:){0,6}
     (?:[0-9]{1,3}\.){3}[0-9]{1,3} # и 4 байта
     (?![:.\w]))                   # и привязка
```

```
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 # Компактная с 7 двоеточиями и 5 числами
|::(?:[A-F0-9]{1,4}:){5}
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
|# Стандартная
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Компактная, не более 7 двоеточий
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # и привязка
 # и не более 1 пары двоеточий
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
 # Компактная с 8 двоеточиями
|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7}
)(?![:.\w])
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3}↵
[0-9]{1,3}(?![:.\w]))(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)↵
|::(?:[A-F0-9]{1,4}:){5})(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|↵
[1-9]?[0-9])\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)|↵
(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}:){0,7}↵
[A-F0-9]{0,4}(?![:.\w]))(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}↵
|:)|(?:[A-F0-9]{1,4}:){7}:|:(:[A-F0-9]{1,4}){7})(?![:.\w])
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Из-за наличия нескольких форм записи сопоставление с адресами IPv6 реализовать не так просто, как сопоставление с адресами IPv4. Выбор допустимых форм записи оказывает большое влияние на сложность регулярного выражения. Обычно используются две формы записи: стандартная и смешанная. Если вы решите считать допустимой только одну из них, то получите три набора регулярных выражений.

И стандартная, и смешанная нотации имеют компактную форму записи, в которой опускаются нулевые значения. Поддержка компактной формы записи дает еще три набора регулярных выражений.

Для проверки, является ли заданная строка допустимым адресом IPv6, и для поиска IP-адресов в текстовом документе потребуются немного отличающиеся регулярные выражения. При проверке допустимости IP-адреса мы использовали якорные метасимволы, описываемые в рецепте 2.5. Диалект JavaScript использует якорные метасимволы ‹^› и ‹$›,

тогда как диалект Ruby использует якорные метасимволы ‹\A› и ‹\Z›. Все остальные диалекты поддерживают оба набора якорных метасимволов. Диалект Ruby также поддерживает метасимволы ‹^› и ‹$›, но допускает их совпадение с символами разрыва строки, присутствующими в тексте. При использовании диалекта Ruby символ крышки и знак доллара должны применяться, только если испытуемый текст не содержит разрывов строк.

Для поиска адресов IPv6 в текстовом документе мы использовали негативную ретроспективную проверку ‹(?<![:.\w])› и негативную опережающую проверку ‹(?![:.\w])›, чтобы убедиться в отсутствии символа слова (буква, цифра или символ подчеркивания), двоеточия или точки перед адресом и после него. Это гарантирует, что совпадение не будет обнаружено в длинной последовательности цифр и двоеточий. Порядок работы ретроспективной и опережающей проверок описывается в рецепте 2.16. Если проверка ближайших символов недоступна, то для проверки отсутствия символа слова перед адресом и после него можно использовать границы слова, но только если первый и последний символы адреса являются (шестнадцатеричной) цифрой. В компактной форме записи адреса могут начинаться и заканчиваться символом двоеточия. В этом случае, чтобы получить совпадение с границей слова перед двоеточием или после него, потребовалось бы наличие соседней буквы или цифры, что совсем не то, что нам требуется. Полное обсуждение границ слова приводится в рецепте 2.6.

Стандартная форма записи

Стандартная форма записи адресов IPv6 достаточно просто поддается обработке с помощью регулярного выражения. В этом случае необходимо обеспечить совпадение с восемью словами в шестнадцатеричной нотации, разделенных двоеточиями. Выражение ‹[A-F0-9]{1,4}› совпадает с последовательностью от 1 до 4 шестнадцатеричных символов, которые составляют 16-битное слово с необязательными ведущими нулями. В символьном классе (рецепт 2.3) перечислены только символы верхнего регистра. Совпадение с символами нижнего регистра обеспечивается за счет режима нечувствительности к регистру символов. Порядок установки режимов в различных языках программирования описывается в рецепте 3.4.

Несохраняющая группа ‹(?:[A-F0-9]{1,4}:){7}› совпадает с шестнадцатеричным словом, за которым следует символ двоеточия. Квантификатор повторяет группу семь раз. Первое двоеточие в этом регулярном выражении является частью синтаксиса определения несохраняющей группы, как описывается в рецепте 2.9, а второй – это литерал двоеточия. Символ двоеточия интерпретируется в регулярных выражениях как метасимвол только в некоторых ситуациях, когда он является частью лексемы регулярного выражения. Поэтому нет необходимости экранировать литерал двоеточия символом обратного слэша. Мы могли бы экранировать его, но это только сделало бы регулярное выражение менее удобочитаемым.

Смешанная форма записи

Регулярное выражение для сопоставления с адресами IPv6 в смешанной форме записи состоит из двух частей. Подвыражение `(?:[A-F0-9]{1,4}:){6}` совпадает с шестью шестнадцатеричными словами, за каждым из которых следует символ двоеточия; точно такое же подвыражение используется для сопоставления с последовательностью из семи шестнадцатеричных слов в стандартной форме записи.

Теперь вместо еще двух шестнадцатеричных слов в конце адреса присутствует полный адрес IPv4. Для сопоставления с ним мы использовали «точное» регулярное выражение из рецепта 8.16, допускающее наличие ведущих нулей.

Стандартная или смешанная форма записи

Чтобы обеспечить совпадение и со стандартной, и со смешанной формами записи, требуется более длинное регулярное выражение. Эти формы записи отличаются только представлением последних 32 битов в адресе IPv6. В стандартной форме записи используются два 16-битных слова, тогда как в смешанной – 4 десятичных байта, как в адресе IPv4.

Первая часть регулярного выражения совпадает с шестью шестнадцатеричными словами, как и в регулярном выражении, поддерживающем только смешанную форму записи. Вторая часть регулярного выражения теперь представлена несохраняющей группой с двумя альтернативами представления последних 32 бит. Как описывается в рецепте 2.8, оператор выбора (вертикальная черта) имеет самый низкий приоритет среди всех операторов регулярных выражений. Поэтому пришлось использовать несохраняющую группировку, чтобы исключить из выбора шесть слов.

Первая альтернатива, слева от вертикальной черты, совпадает с двумя шестнадцатеричными словами и символом двоеточия между ними. Вторая альтернатива совпадает с адресом IPv4.

Компактная форма записи

Когда необходимо обеспечить совпадение с компактной формой записи, дело осложняется еще больше. Причина в том, что компактная форма записи позволяет опускать переменное число нулей. Так, `1:0:0:0:0:6:0:0`, `1::6:0:0` и `1:0:0:0:0:6::` – это три разных способа записи одного и того же адреса IPv6. Адрес может содержать до восьми слов, но может и не содержать ни одного. Если в адресе присутствует менее восьми слов, в нем должна иметься одна последовательность из двух символов двоеточия, представляющая опущенные нулевые значения.

Переменное число повторений легко реализовать в регулярных выражениях. Если в адресе IPv6 имеется пара двоеточий, следующих друг за другом, то в адресе может присутствовать не более семи слов перед и после двойного двоеточия. Это легко записать следующим образом:

```
(
  ([0-9A-F]{1,4}:){1,7} # от 1 до 7 слов слева
| :                     # или пара двоеточий в начале
)
(
  (:[0-9A-F]{1,4}){1,7} # от 1 до 7 слов слева
| :                     # или пара двоеточий в конце
)
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Это регулярное выражение, как и следующие ниже в этом разделе, также будет работать и в диалекте JavaScript, если убрать из него комментарии и лишние пробельные символы. Диалект JavaScript поддерживает все функциональные возможности, задействованные в этих регулярных выражениях, за исключением режима свободного форматирования, который мы используем здесь, чтобы упростить понимание регулярных выражений. Или можно использовать библиотеку XRegExp, которая наряду с другими дополнительными особенностями поддерживает режим свободного форматирования регулярных выражений в JavaScript.

Это регулярное выражение совпадает со всеми адресами IPv6 в компактной форме записи, но оно не будет совпадать с адресами в стандартной, некомпактной форме.

Это очень простое регулярное выражение. Первая его часть совпадает с последовательностью от 1 до 7 слов, за которой следует символ двоеточия, или с двоеточием для адресов, в которых нет ни одного слова левее двойного двоеточия. Вторая часть совпадает с последовательностью от 1 до 7 слов, которой предшествует символ двоеточия, или с двоеточием для адресов, в которых нет ни одного слова правее двойного двоеточия. Все вместе обеспечивает допустимость совпадения с самим двоеточием, с двоеточием и последовательностью от 1 до 7 слов только слева, с двоеточием и последовательностью от 1 до 7 слов только справа и с двоеточием и последовательностью от 1 до 7 слов слева и справа.

Эта последняя часть доставляет определенные неприятности. Регулярное выражение допускает совпадение с последовательностью от 1 до 7 слов как слева, так и справа, как собственно и должно быть, но оно не позволяет указать, что суммарное количество слов слева и справа не должно превышать 7. В адресе IPv6 должно быть 8 слов. Двойное двоеточие свидетельствует, что было опущено по крайней мере одно слово, поэтому общее число слов не должно превышать 7.

Регулярные выражения не производят математических вычислений. Они могут ограничить появление некоторого фрагмента от 1 до 7 раз. Но они не в состоянии ограничить суммарное появление двух фрагмен-

тов, распределив эти 7 раз между двумя фрагментами в какой-нибудь пропорции.

Чтобы лучше понять эту проблему, рассмотрим простую аналогию. Допустим, что необходимо обеспечить совпадение с чем-то, что записывается в формате aaaaxbbb. Строка должна иметь длину от 1 до 8 символов и содержать от 0 до 7 символов a, точно один символ x и от 0 до 7 символов b.

Решить эту проблему с помощью регулярных выражений можно двумя способами. Один из них заключается в том, чтобы описать все возможные альтернативы. Этот способ используется в следующем разделе, где обсуждается сопоставление с компактной смешанной формой записи. Это может привести к созданию весьма длинного регулярного выражения, но его легко понять.

```
\A(?:a{7}x
 |  a{6}xb?
 |  a{5}xb{0,2}
 |  a{4}xb{0,3}
 |  a{3}xb{0,4}
 |  a{2}xb{0,5}
 |  axb{0,6}
 |  xb{0,7}
)\Z
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В этом регулярном выражении для каждого возможного числа символов a предусмотрена своя альтернатива. Каждая альтернатива описывает допустимое количество символов b, после того как было найдено совпадение с заданным числом символов a и символом x.

Другое решение состоит в том, чтобы использовать опережающую проверку. Этот метод используется в регулярном выражении в разделе «Решение», совпадающем с адресом IPv6 в компактной форме записи. Те, кто не знаком с опережающей проверкой, могут сначала ознакомиться с рецептом 2.16. Опережающая проверка обеспечивает возможность сопоставления с одним и тем же фрагментом текста дважды с проверкой соблюдения разных условий.

```
\A
  (?=[abx]{1,8}\Z)
  a{0,7}xb{0,7}
\Z
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Якорный метасимвол ‹\A› в начале регулярного выражения привязывает его к началу испытуемого текста. Далее следует позитивная опережающая проверка. Она проверяет, имеется ли совпадение с последова-

тельностью от 1 до 8 букв ‹a›, ‹b› и/или ‹x›, и находится ли это совпадение в конце текста. Метасимвол ‹\Z› внутри опережающей проверки имеет крайне важное значение. Чтобы регулярное выражение могло ограничить длину строки восемью символами, опережающая проверка должна убедиться в отсутствии дополнительных символов после совпавших.

В другой ситуации вместо ‹\A› и ‹\Z› можно было бы использовать другие символы-ограничители. Если необходимо было бы выполнить поиск последовательности aaaaxbbb и ее разновидностей «только как целых слов», можно было бы использовать границы слов. Но в любом случае, чтобы ограничить длину совпадения, необходимо использовать какой-то ограничитель, и этот ограничитель, совпадающий с концом строки, необходимо поместить и в опережающую проверку, и в конец регулярного выражения. Если этого не сделать, регулярное выражение будет совпадать с частью более длинной строки.

Когда опережающая проверка удовлетворит свои требования, она вернет обратно совпавшие с ней символы. То есть, когда механизм регулярных выражений перейдет к подвыражению ‹a{0,7}›, он окажется в начале испытуемого текста. Тот факт, что опережающая проверка не поглощает текст, совпавший с ней, является главным ее отличием от несохраняющей группы и позволяет применять два шаблона к одному и тому же фрагменту текста.

Несмотря на то что подвыражение ‹a{0,7}xb{0,7}› само по себе могло бы совпасть с последовательностью длиной до 15 символов, тем не менее в данном случае оно может совпасть только с 8-ю, потому что опережающая проверка уже гарантирует наличие всего 8 символов. Все, что остается на долю подвыражения ‹a{0,7}xb{0,7}›, – это проверка правильного порядка следования символов. При этом подвыражение ‹a*xb*› могло бы обеспечить тот же эффект, что и подвыражение ‹a{0,7}xb{0,7}›.

Второй метасимвол ‹\Z› в конце регулярного выражения тоже имеет важное значение. Если в опережающей проверке необходимо проверить отсутствие лишних символов, то во второй проверке необходимо гарантировать верный порядок следования символов. Это дает уверенность, что регулярное выражение не совпадет с чем-нибудь вроде axba, когда длина текста находится в диапазоне от 1 до 8 символов и удовлетворяет требованиям опережающей проверки.

Компактная смешанная форма записи

Смешанную форму записи, так же как и стандартную, можно сжать. Несмотря на то что четыре байта в конце должны быть указаны всегда, даже когда они равны нулю, количество шестнадцатеричных слов перед ними может быть переменным. Если все шестнадцатеричные слова равны нулю, адрес IPv6 может в конечном итоге выглядеть, как адрес IPv4, которому предшествуют два двоеточия.

При создании регулярного выражения, совпадающего с адресами в компактной смешанной форме записи, приходится решать те же проблемы, что и в случае с компактной стандартной формой записи. Все они были описаны в предыдущем разделе.

Главное отличие между регулярными выражениями для сопоставления с компактной смешанной и с компактной (стандартной) формами записи заключается в том, что при сопоставлении с адресами в компактной смешанной нотации необходимо проверять наличие адреса IPv4 после шести шестнадцатеричных слов. Эта проверка выполняется в конце регулярного выражения с помощью подвыражения из рецепта 8.16, обнаруживающего точное совпадение с адресами IPv4 в некомпактной смешанной форме записи.

Проверка совпадения с адресом IPv4 в конце выражения необходима, но точно так же она необходима и внутри опережающей проверки, чтобы убедиться, что в адресе IPv6 присутствует не более шести двоеточий или шести шестнадцатеричных слов. Так как точное сопоставление уже выполняется в конце регулярного выражения, в опережающей проверке вполне достаточно простой проверки совпадения с адресом IPv4. Точная проверка адреса IPv4 в опережающей проверке не требуется, так как это делает основное регулярное выражение. Но она должна обеспечивать совпадение с частью IPv4, чтобы якорный символ, совпадающий с концом текста внутри опережающей проверки, мог выполнить свою работу.

Стандартная, смешанная или компактная форма записи

Заключительный набор регулярных выражений объединяет в себе все вышесказанное. Они совпадают с адресами IPv6 в любой форме записи: стандартной или смешанной, компактной или нет.

Регулярные выражения в этом наборе составлены из выражений для компактной смешанной и компактной (стандартной) форм записи с применением конструкции выбора. В них, в свою очередь, также используются конструкции выбора для сопоставления с компактной и некомпактной формами записи адресов IPv6.

В результате получилось регулярное выражение с тремя конструкциями выбора на верхнем уровне, первая из которых в свою очередь содержит два варианта. Первая конструкция соответствует адресам IPv6 в смешанной форме записи, как компактной, так и некомпактной. Вторая конструкция соответствует адресам IPv6 в стандартной форме записи. Третья конструкция соответствует адресам в компактной (стандартной) форме записи.

Мы создали три конструкции выбора верхнего уровня вместо двух со своими собственными конструкциями выбора, потому что нет никаких причин группировать в конструкцию выбора стандартную и компактную формы записи. В случае смешанной формы записи необходимость

объединения компактной и некомпактной форм записи обусловлена возможностью сэкономить на проверке части адреса IPv4.

Фактически мы объединили выражение:

```
^(6words|compressed6words)ip4$
```

и выражение:

```
^(8words|compressed8words)$
```

в выражение:

```
^((6words|compressed6words)ip4|8words|compressed8words)$
```

вместо:

```
^((6words|compressed6words)ip4|(8words|compressed8words))$
```

См. также

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках. В рецепте 2.18 описывается, как добавлять комментарии.

8.18. Проверка путей в Windows

Задача

Необходимо проверить, является ли некоторая строка допустимой строкой пути к папке или к файлу в операционной системе Microsoft Windows.

Решение

Буква устройства в пути

```
\A
[a-z]:\\                    # Устройство
(?:[^\\/:*?"<>|\r\n]+\\)*   # Папка
[^\\/:*?"<>|\r\n]*          # Файл
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z]:\\(?:[^\\/:*?"<>|\r\n]+\\)*[^\\/:*?"<>|\r\n]*$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Буква устройства и пути в формате UNC

```
\A
(?:[a-z]:|\\\\[a-z0-9_.$\-]+\\[a-z0-9_.$\-]+)\\ # Устройство
(?:[^\\/:*?"<>|\r\n]+\\)*                       # Папка
[^\\/:*?"<>|\r\n]*                              # Файл
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:[a-z]:|\\\\[a-z0-9_.$-]+\\[a-z0-9_.$-]+)\\(?:[^\\/:*?"<>|\r\n]+\\)*↵
[^\\/:*?"<>|\r\n]*$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Буква устройства, UNC и относительные пути

```
\A
?:(?:[a-z]:|\\\\[a-z0-9_.$\-]+\\[a-z0-9_.$\-]+)\\| # Устройство
  \\?[^\\/:*?"<>|\r\n]+\\?)                        # Относительный путь
(?:[^\\/:*?"<>|\r\n]+\\)*                          # Папка
[^\\/:*?"<>|\r\n]*                                 # Файл
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:[a-z]:|\\\\[a-z0-9_.$-]+\\[a-z0-9_.$-]+)\\|\\?[^\\/:*?"<>|↵
\r\n]+\\?)(?:[^\\/:*?"<>|\r\n]+\\)*[^\\/:*?"<>|\r\n]*$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Буква устройства в пути

Сопоставление с полным путем к файлу или папке на устройстве, в котором присутствует буква устройства, выполняется очень просто. Устройство обозначается одной буквой, за которой следуют двоеточие и обратный слэш. Это совпадение обеспечивает простое выражение ‹[a-z]:\\›. Обратный слэш является метасимволом в регулярных выражениях, поэтому, чтобы обеспечить совпадение с литералом, его необходимо экранировать другим обратным слэшем.

Имена папок и файлов в Windows могут содержать любые символы, за исключением следующих: \/:*?"<>|. Разрывы строк в именах недопустимы. Мы легко можем описать совпадение с любым символом, за исключением указанных, с помощью инвертированного символьного класса ‹[^\\/:*?"<>|\r\n]+›. Обратный слэш в символьных классах также является метасимволом, поэтому его необходимо экранировать. Последовательности ‹\r› и ‹\n› – это два символа разрыва строки. Подробнее о символьных классах рассказывается в рецепте 2.3. Квантификатор «плюс» (рецепт 2.12) указывает, что требуется совпадение с одним или более символами.

Имена папок в пути отделяются друг от друга обратными слэшами. Совпадение с последовательностью из нуля или более имен папок можно описать с помощью выражения ‹(?:[^\\/:*?"<>|\r\n]+\\)*›, которое помещает выражение, соответствующее имени папки, и литерал обратного слэша в несохраняющую группу (рецепт 2.9), повторяемую ноль или более раз с помощью звездочки (рецепт 2.12).

Для сопоставления с именем файла используется выражение ‹[^\\/:*?"<>|\r\n]*›. Звездочка обеспечивает необязательность имени файла, благодаря чему допускается завершение пути символом обратного слэша. Если необходимо, чтобы обратный слэш отсутствовал в конце пути, нужно заменить последний символ ‹*› в регулярном выражении на ‹+›.

Буква устройства и пути в формате UNC

Пути к файлам на сетевых устройствах, которые не отображаются на буквы устройств, можно записывать в соответствии с универсальным соглашением об именах (Universal Naming Convention, UNC). Пути в формате UNC имеют вид *\\server\share\folder\file*.

Мы легко можем адаптировать регулярное выражение, совпадающее с путями, включающими букву устройства, для сопоставления с путями в формате UNC. Для этого достаточно заменить подвыражение ‹[a-z]:›, совпадающее с буквой устройства, на подвыражение, совпадающее с буквой устройства или с именем сервера.

Такое совпадение обеспечивает выражение ‹(?:[a-z]:|\\\\[a-z0-9_.$•-]+\\[a-z0-9_.$•-]+)›. Вертикальная черта – это оператор выбора (рецепт 2.8). Он реализует выбор между сопоставлением ‹[a-z]:› с буквой устройства и сопоставлением ‹\\\\[a-z0-9_.$•-]+\\[az0-9_.$•-]+› с именем сервера и разделяемого ресурса. Оператор выбора имеет самый низкий приоритет среди всех операторов регулярных выражений. Чтобы объединить два варианта, мы использовали несохраняющую группировку. Как объясняется в рецепте 2.9, символы ‹(?:› образуют сложного вида открывающую скобку несохраняющей группы. Вопросительный знак после открывающей круглой скобки теряет свой обычный смысл.

Остальная часть регулярного выражения может оставаться той же самой. Имя разделяемого ресурса в формате UNC будет совпадать с частью регулярного выражения, которая соответствует именам папок.

Буква устройства, UNC и относительные пути

Относительный путь начинается с имени папки (возможно, со специального имени папки .., соответствующего родительской папке) или содержит только имя файла. Чтобы обеспечить поддержку относительных путей, необходимо добавить третью конструкцию в раздел регулярного выражения, совпадающего с «устройством». Эта конструкция должна совпадать с началом относительного пути, а не с буквой устройства или именем сервера.

Выражению ‹\\?[^\\/:*?"<>|\r\n]+\\?› соответствует начало относительного пути. Путь может начинаться с символа обратного слэша, но это необязательно. Подвыражение ‹\\?› совпадет с символом обратного слэша, если он присутствует, или ни с чем в противном случае. Подвыражение ‹[^\\/:*?"<>|\r\n]+› совпадает с именем папки или файла. Если относительный путь содержит только имя файла, заключительное подвыражение ‹\\?› не будет совпадать ни с чем, так же как и обе части регулярного выражения, соответствующие «папке» или «файлу», каждая из которых является необязательной. Если относительный путь определяет каталог, заключительное подвыражение ‹\\?› будет совпадать с обратным слэшем, отделяющим первую папку в относительном пути от остальной части пути. Часть регулярного выражения, соответствующая «папке», в этом случае совпадет с остальными именами папок в пути, если они присутствуют, а часть регулярного выражения, соответствующая «файлу», совпадет с именем файла.

Регулярное выражение, соответствующее относительному пути, уже не имеет отдельных частей, соответствующих различным компонентам испытуемого текста. Часть регулярного выражения, помеченная как «относительный путь», фактически будет совпадать с именем папки или файла, если путь относительный. Если в относительном пути присутствует одно или более имен папок, часть регулярного выражения «относительный путь» совпадет с первой папкой, а части «папка» и «файл» совпадут с тем, что останется. Если относительный путь состоит только из имени файла, ему будет соответствовать часть «относительный путь», а на долю частей «папка» и «файл» ничего не останется. Так как нас интересует только проверка корректности пути, это обстоятельство не имеет никакого значения. Комментарии в регулярном выражении играют роль меток, чтобы помочь вам лучше понять его.

Если потребуется извлекать отдельные части пути в сохраняющие группы, то придется приложить больше усилий, чтобы обеспечить совпадение с буквой устройства, именем папки или файла по отдельности. Эта проблема решается в следующем рецепте.

См. также

Рецепт 8.19, в котором также проверяются пути Windows, а кроме того, добавляются сохраняющие группы для буквы устройства, папки и файла, позволяя извлекать их по отдельности.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.18 описывается, как добавлять комментарии.

8.19. Выделение элементов путей в Windows

Задача

Необходимо проверить, выглядит ли испытуемый текст как допустимый путь к папке или к файлу в операционной системе Microsoft Windows. Если окажется, что испытуемый текст хранит допустимый путь Windows, то необходимо извлечь букву устройства, часть пути с именами папок и имя файла по отдельности.

Решение

Буква устройства в пути

```
\A
(?<drive>[a-z]:)\\
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java 7, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:)\\
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

```
\A
([a-z]:)\\
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z]:)\\((?:[^\\/:*?"<>|\r\n]+\\)*)([^\\/:*?"<>|\r\n]*)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Буква устройства и пути в формате UNC

```
\A
(?<drive>[a-z]:|\\\\[a-z0-9_.$•-]+\\[a-z0-9_.$•-]+)\\
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java 7, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:|\\\\[a-z0-9_.$•-]+\\[a-z0-9_.$•-]+)\\
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

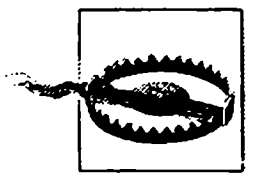
```
\A
([a-z]:|\\\\[a-z0-9_.$•-]+\\[a-z0-9_.$•-]+)\\
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z]:|\\\\[a-z0-9_.$•-]+\\[a-z0-9_.$•-]+)\\((?:[^\\/:*?"<>|\r\n]+\\)*)↵
([^\\/:*?"<>|\r\n]*)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Буква устройства, UNC и относительные пути



Следующие регулярные выражения могут совпадать с пустой строкой. Дополнительные сведения и альтернативное решение приводятся в разделе «Обсуждение».

```
\A
(?<drive>[a-z]:\\|\\\\[a-z0-9_.$•-]+\\[a-z0-9_.$•-]+\\|\\?)
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java 7, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:\\|\\\\[a-z0-9_.$%-]+\\[a-z0-9_.$%-]+\\|\\?)
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

```
\A
([a-z]:\\|\\\\[a-z0-9_.$%-]+\\[a-z0-9_.$%-]+\\|\\?)
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z]:\\|\\\\[a-z0-9_.$%-]+\\[a-z0-9_.$%-]+\\|\\?)↵
((?:[^\\/:*?"<>|\r\n]+\\)*)([^\\/:*?"<>|\r\n]*)$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Обсуждение

Регулярные выражения в этом рецепте очень похожи на выражения в предыдущем рецепте. Здесь предполагается, что вы уже прочитали и поняли пояснения, которые приводились в предыдущем рецепте.

Буква устройства в пути

В регулярное выражение, совпадающее со строкой пути, включающей букву устройства, было внесено единственное изменение по сравнению с выражением из предыдущего рецепта. Мы добавили три сохраняющие группы, которые можно использовать для извлечения различных частей пути: `<drive>` (устройство), `<folder>` (папка) и `<file>` (файл). Если используемый диалект регулярных выражений поддерживает именованные сохранения (рецепт 2.11), можно использовать эти имена. В противном случае можно ссылаться на сохраняющие группы по их номерам: 1, 2 и 3. В рецепте 3.9 можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Буква устройства и пути в формате UNC

В выражение, совпадающее с путями в формате UNC, мы добавили те же три сохраняющие группы.

Буква устройства, UNC и относительные пути

Регулярное выражение становится сложнее, когда возникает необходимость обеспечить совпадение с относительными путями. В предыдущем рецепте просто была добавлена третья конструкция в часть выражения, совпадающую с началом относительного пути. В данном случае это невозможно. При совпадении с относительными путями сохраняющая группа, соответствующая букве устройства, должна оставаться пустой.

На этот раз символ обратного слэша, следовавший в выражении из раздела «Буква устройства и пути в формате UNC» вслед за сохраняющей группой с буквой устройства, переместился в эту сохраняющую группу. Мы добавили его в конец конструкций выбора, соответствующих букве устройства и сетевому имени разделяемого ресурса. Мы также добавили третью конструкцию выбора с необязательным обратным слэшем для совпадения с относительными путями, которые могут начинаться с символа обратного слэша. Так как третья конструкция является необязательной, то и вся группа, совпадающая с устройством, фактически становится необязательной.

Получившееся регулярное выражение будет совпадать с любыми путями в операционной системе Windows. Проблема состоит в том, что сделав необязательной часть, совпадающую с устройством, мы получили регулярное выражение, в котором все части являются необязательными. Части, совпадающие с именами папок и файлов, уже были необязательными в выражениях, поддерживавших только абсолютные пути. Другими словами, это регулярное выражение может совпадать с пустой строкой.

Если необходимо гарантировать невозможность совпадения с пустой строкой, следует добавить дополнительные альтернативы для работы с относительными путями, которые определяют папку (когда имя файла является необязательным), и с относительными путями, которые не определяют папку (когда имя файла является обязательным):

```
\A
(?:
  (?<drive>[a-z]:|\\\\[a-z0-9_.$•-]+\\[a-z0-9_.$•-]+)\\
  (?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<relativefolder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?<file2>[^\\/:*?"<>|\r\n]*)
| (?<relativefile>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: .NET, Java 7, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?:
```

```
  (?P<drive>[a-z]:|\\\\[a-z0-9_.$%-]+\\[a-z0-9_.$%-]+)\\
  (?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?P<file>[^\\/:*?"<>|\r\n]*)
| (?P<relativefolder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?P<file2>[^\\/:*?"<>|\r\n]*)
| (?P<relativefile>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: PCRE 4 и более поздние версии, Perl 5.10, Python

```
\A
(?:
  ([a-z]:|\\\\[a-z0-9_.$%-]+\\[a-z0-9_.$%-]+)\\
  ((?:[^\\/:*?"<>|\r\n]+\\)*)
  ([^\\/:*?"<>|\r\n]*)
| (\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  ([^\\/:*?"<>|\r\n]*)
| ([^\\/:*?"<>|\r\n]+)
)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов

Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:([a-z]:|\\\\[a-z0-9_.$%-]+\\[a-z0-9_.$%-]+)\\↵
((?:[^\\/:*?"<>|\r\n]+\\)*)([^\\/:*?"<>|\r\n]*)|(\\?(?:[^\\/:*?"<>|↵
\r\n]+\\)+)([^\\/:*?"<>|\r\n]*)|([^\\/:*?"<>|\r\n]+))$
```

Параметры: нечувствительность к регистру символов

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

Из-за того что была исключена возможность совпадения с пустой строкой, мы получили шесть сохраняющих групп, которые сохраняют три различные части пути. В каждом конкретном случае следует определиться заранее, что будет проще – выполнить дополнительную проверку на отсутствие символов в строке перед применением этого выражения или приложить дополнительные усилия для обработки нескольких сохраняющих групп после того, как совпадение уже будет найдено. Диалекты Perl 5.10, Ruby 1.9 и .NET допускают возможность присваивания одного и того же имени нескольким именованным группам. Подробнее об этом рассказывается в рецепте 2.11 в разделе «Группы с одинаковыми именами». Благодаря этому можно просто извлекать совпадение с группой ‹folder› (папка) или ‹file› (файл), не беспокоясь о том, какая из двух групп ‹folder› или какая из трех групп ‹file› фактически участвовала в совпадении:

```
\A
(?:
  (?<drive>[a-z]:|\\\\[a-z0-9_.$%-]+\\[a-z0-9_.$%-]+)\\
```

```
  (?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<folder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<file>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Параметры: режим свободного форматирования, нечувствительность к регистру символов
Диалекты: NET, Perl 5.10, Ruby 1.9

См. также

Рецепт 8.18, в котором также проверяются пути Windows с использованием упрощенных регулярных выражений без отдельных сохраняющих групп для устройства, папки и файла.

В рецепте 3.9 описывается, как извлекать фрагменты текста, совпавшие с сохраняющими группами в программном коде. Воспользуйтесь им, чтобы реализовать извлечение необходимых частей пути.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.11 обсуждаются именованные группировки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

8.20. Извлечение буквы устройства из путей в Windows

Задача

Имеется строка, в которой хранится синтаксически допустимый путь к файлу или папке на диске в системе Windows или в сети. Необходимо извлечь букву устройства, если таковая присутствует. Например, необходимо извлечь букву `c` из пути `c:\folder\file.ext`.

Решение

```
^([a-z]):
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение буквы устройства из строки, которая заведомо содержит допустимый путь, является тривиальной задачей, даже если заранее не известно, действительно ли путь начинается с буквы устройства. Путь может быть или относительным, или в формате UNC.

Двоеточие в строках путей в системе Windows является недопустимым символом, за исключением случая, когда оно используется для отделения буквы устройства. То есть если в начале строки имеется буква, за которой следует двоеточие, можно быть уверенными, что эта буква является буквой устройства.

Якорный метасимвол `^` совпадает с началом строки (рецепт 2.5). Тот факт, что в Ruby символ крышки также совпадает с разрывами строк, не имеет большого значения, потому что допустимые пути в Windows не содержат разрывов строк. Символьный класс `[a-z]` совпадает с единственной буквой (рецепт 2.3). Мы поместили символьный класс между парой круглых скобок (образующих сохраняющую группу), благодаря чему получили возможность извлекать букву устройства без символа двоеточия, который также совпадает с регулярным выражением. Мы добавили двоеточие в регулярное выражение, чтобы гарантировать совпадение с буквой устройства, а не с первой буквой в относительном пути.

См. также

Рецепт 2.9, где рассказывается о сохраняющей группировке.

Рецепт 3.9, где можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в том или ином языке программирования.

Рецепт 8.19, если заранее неизвестно, хранит ли испытуемая строка допустимый путь Windows.

8.21. Извлечение имен сервера и разделяемого ресурса из пути в формате UNC

Задача

Имеется строка, в которой хранится синтаксически допустимый путь к файлу или папке на диске в системе Windows или в сети. Если путь указан в формате UNC, необходимо извлечь имя сетевого сервера и имя разделяемого ресурса на сервере, содержащиеся в пути. Например, необходимо извлечь server и share из пути `\\server\share\folder\file.ext`.

Решение

```
^\\\\([a-z0-9_.$-]+)\\([a-z0-9_.$-]+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение имени сетевого сервера и разделяемого ресурса из строки, которая заведомо содержит допустимый путь, является простой задачей, даже если заранее неизвестно, действительно ли строка содержит путь в формате UNC. Путь может быть относительным или начинаться с буквы устройства.

Пути в формате UNC начинаются с двух символов обратного слэша. Два обратных слэша в строках путей в системе Windows являются недопустимыми, за исключением случая, когда они стоят в начале пути в формате UNC. То есть если в начале строки имеются два символа обратного слэша, можно быть уверенными, что за ними следуют имя сервера и имя разделяемого ресурса.

Якорный метасимвол ‹^› совпадает с началом строки (рецепт 2.5). Тот факт, что в диалекте Ruby символ крышки также совпадает с разрывами строк, не имеет большого значения, потому что допустимые пути в Windows не содержат разрывов строк. Подвыражение ‹\\\\› совпадает с двумя литералами обратного слэша. Так как обратный слэш в регулярных выражениях является метасимволом, его необходимо экранировать другим символом обратного слэша, чтобы обеспечить совпадение с литералом. Первый символьный класс ‹[a-z0-9_.$*-]+› совпадает с именем сетевого сервера. Второй, следующий за другим литералом обратного слэша, совпадает с именем разделяемого ресурса. Мы поместили символьные классы между парами круглых скобок, образующими сохраняющие группы, чтобы иметь возможность извлекать имя сервера из первой сохраняющей группы, а имя разделяемого ресурса – из второй. Все регулярное выражение будет совпадать с фрагментом \\server\share.

См. также

Рецепт 2.9, где рассказывается о сохраняющей группировке.

Рецепт 3.9, где можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в том или ином языке программирования.

Рецепт 8.19, если заранее неизвестно, хранит ли испытуемая строка допустимый путь Windows.

8.22. Извлечение имен папок из путей в Windows

Задача

Имеется строка, в которой хранится синтаксически допустимый путь к файлу или папке на диске в системе Windows или в сети. Необходимо извлечь из пути имя папки. Например, необходимо извлечь фрагмент \folder\subfolder\ из пути c:\folder\subfolder\file.ext или из пути \\server\share\folder\subfolder\file.ext.

Решение

```
^([a-z]:|\\\\[a-z0-9_.$#-]+\\[a-z0-9_.$#-]+)?((?:\\|^)↵
(?:[^\\/:*?"<>|\r\n]+\\)+)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение имени папки из строки, содержащей путь в Windows, может оказаться непростым делом, если требуется обеспечить поддержку формата UNC, потому что нельзя просто извлечь часть пути между символами обратного слэша. В этом случае в совпадение попадут имя сетевого сервера и имя разделяемого ресурса.

Первая часть регулярного выражения, `‹^([a-z]:|\\\\[a-z0-9_.$#-]+\\[a-z0-9_.$#-]+)?›`, перепрыгивает через букву устройства или имя сетевого сервера и имя разделяемого ресурса, находящиеся в начале пути. Эта часть регулярного выражения состоит из сохраняющей группы с двумя альтернативами. Первая совпадает с буквой устройства, как описывается в рецепте 8.20, а вторая – с именем сервера и именем разделяемого ресурса в пути, представленном в формате UNC, как описывается в рецепте 8.21. Оператор выбора описывается в рецепте 2.8.

Знак вопроса, следующий за группой, делает ее необязательной. Это позволяет обеспечить поддержку относительных путей, в которых отсутствует буква устройства или имя разделяемого сетевого ресурса.

Совпадение с папками легко обеспечить с помощью подвыражения `‹(?:[^\\/:*?"<>|\r\n]+\\)+›`. Символьный класс совпадает с именем папки. Несохраняющая группа совпадает с именем папки, за которым следует символ обратного слэша, отделяющий папки друг от друга и от имени файла. Группа повторяется один или более раз. Это означает, что данное регулярное выражение будет совпадать только с теми путями, в которых действительно присутствует имя папки. Пути, в которых указывается только имя файла, устройство или имя сетевого ресурса, совпадать с регулярным выражением не будут.

Если путь начинается с буквы устройства или с имени сетевого ресурса, за ними должен следовать символ обратного слэша. Относительные пути могут начинаться с обратного слэша. Поэтому в начало группы в части выражения, совпадающей с папкой, необходимо добавить необязательный обратный слэш. Так как данное регулярное выражение предполагается использовать только с заведомо допустимыми путями, нет необходимости требовать наличия обратного слэша в случае наличия в пути буквы устройства или имени сетевого ресурса. Достаточно лишь допустить его присутствие.

Регулярное выражение должно совпадать по меньшей мере с одной папкой, поэтому мы обеспечили невозможность совпадения с фрагментом

e\ как с именем папки в пути \\server\share\. Кроме того, чтобы добавить совпадение с необязательным обратным слэшем в начало сохраняющей группы, совпадающей с именем папки, мы использовали подвыражение ‹(\\|^)›, а не ‹\\?›.

Если вам интересно узнать, почему фрагмент \\server\shar может совпадать как буква устройства, а фрагмент e\ – как имя папки, загляните в рецепт 2.13. Это обусловлено тем, что механизмы регулярных выражений выполняют возвраты. Рассмотрим следующее регулярное выражение:

```
^([a-z]:|\\\\\[a-z0-9_.$*-]+\\[a-z0-9_.$*-]+)?↵
((?:\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
```

Это регулярное выражение, так же как и регулярное выражение, представленное в решении, требует наличия в пути хотя бы одного символа, отличного от обратного слэша, и одного обратного слэша. Если регулярное выражение обнаружит в пути \\server\share совпадение \\server\share с буквой устройства и затем потерпит неудачу при сопоставлении группы, совпадающей с именем папки, оно не остановится на достигнутом, а попытается выполнить различные перестановки.

В этом случае механизм вспомнит, что символьный класс ‹[a-z0-9_.$*-]+›, соответствующий имени сетевого ресурса, необязательно должен соответствовать всем доступным символам. Чтобы удовлетворить квантификатор ‹+›, достаточно одного символа. Механизм выполнит возврат, заставив символьный класс уступить один символ, и повторит попытку.

Когда механизм продолжит сопоставление, в испытуемой строке останется два символа, способных обеспечить совпадение с папкой: e\. Этих двух символов вполне достаточно, чтобы удовлетворить подвыражение ‹(?:[^\\/:*?"<>|\r\n]+\\)+›, в результате чего будет обеспечено совпадение для всего регулярного выражения. Но это не то совпадение, которое ожидалось.

Использовав подвыражение ‹(\\|^)› вместо ‹\\?›, мы решили эту проблему. Оно также допускает наличие обратного слэша, но в случае его отсутствия оно требует, чтобы в начале строки находилось имя папки. Это означает, что если будет обнаружено совпадение с буквой устройства и механизм регулярных выражений таким образом продвинется дальше начала строки, наличие обратного слэша становится обязательным. Если далее не будет найдено совпадение ни с одной папкой, механизм регулярных выражений также попытается выполнить возврат, но безуспешно, потому что сопоставление с подвыражением ‹(\\|^)› потерпит неудачу. Механизм регулярных выражений будет производить возвраты, пока не достигнет начала строки. Сохраняющая группа, совпадающая с буквой устройства и именем сетевого ресурса, является необязательной, поэтому механизм регулярных выражений сможет попытаться выполнить сопоставление для части выражения, совпадающей с име-

нем папки, с начала строки. Хотя подвыражение ‹(\\|^)› сможет обнаружить там совпадение, остальная часть регулярного выражения не допустит этого, потому что подвыражение ‹(?:[^\\/:*?"<>|\r\n]+\\)+› делает невозможным совпадение с двоеточием, следующим за буквой устройства, или с двойным обратным слэшем, с которого начинается имя сетевого ресурса.

Мы не использовали этот прием в рецептах 8.18 и 8.19, потому что в тех регулярных выражениях совпадение с папкой было необязательным. Так как в тех регулярных выражениях все, что следовало за частью, совпадающей с устройством, объявлено необязательным, механизм регулярных выражений никогда не будет выполнять никаких возвратов. Конечно, такая необязательность может привести к другим проблемам, как описывается в рецепте 8.19.

Когда регулярное выражение обнаружит совпадение, первая сохраняющая группа будет содержать букву устройства или имя сетевого ресурса, а вторая сохраняющая группа – имя папки. При совпадении с относительным путем первая сохраняющая группа будет содержать пустую строку. Вторая сохраняющая группа всегда будет содержать хотя бы одно имя папки. Если применить это регулярное выражение к пути, в котором папка отсутствует, оно вообще не будет обнаруживать совпадение.

См. также

Рецепт 2.9, где рассказывается о сохраняющей группировке.

Рецепт 3.9, где можно узнать, как извлекать текст, совпавший с сохраняющей группировкой, в различных языках программирования.

Рецепт 8.19, если заранее неизвестно, хранит ли испытуемая строка допустимый путь Windows.

8.23. Извлечение имени файла из пути Windows

Задача

Имеется строка, в которой хранится синтаксически допустимый путь к файлу или папке на диске в системе Windows или в сети. Необходимо извлечь из пути имя файла. Например, необходимо извлечь фрагмент `file.ext` из пути `c:\folder\file.ext`.

Решение

```
[^\\/:*?"<>|\r\n]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Извлечение имени файла из строки, которая заведомо содержит допустимый путь, является тривиальной задачей, даже если заранее неизвестно, действительно ли путь оканчивается именем файла.

Имя файла всегда находится в конце строки. Оно не может содержать символы двоеточия или обратного слэша, поэтому его невозможно перепутать с именем папки, с буквой устройства или с именем сетевого ресурса, в которых используются символы обратного слэша и/или двоеточия.

Якорный метасимвол ‹$› совпадает с концом строки (рецепт 2.5). Тот факт, что в диалекте Ruby знак доллара также совпадает с разрывами строк, не имеет большого значения, потому что допустимые пути в Windows не содержат разрывов строк. Инвертированный символьный класс ‹[^\\/:*?"<>|\r\n]+› (рецепт 2.3) совпадает с символами, которые могут присутствовать в именах файлов. Несмотря на то что механизм регулярных выражений сканирует испытуемую строку в направлении слева направо, якорь в конце регулярного выражения гарантирует, что только совпадение с последними символами в строке будет интерпретироваться как имя файла.

Если испытуемая строка оканчивается обратным слэшем, как в путях, где не указываются имена файлов, регулярное выражение вообще не будет обнаруживать совпадение. Когда оно действительно будет обнаруживать соответствие, совпадение будет содержать только имя файла, поэтому нет необходимости в использовании сохраняющей группы для отделения имени файла от остальной части пути.

См. также

Рецепт 3.7, где можно узнать, как извлекать текст, совпавший с регулярным выражением, в различных языках программирования.

Рецепт 8.19, если заранее неизвестно, хранит ли испытуемая строка допустимый путь Windows.

8.24. Извлечение расширения имени файла из пути Windows

Задача

Имеется строка, в которой хранится синтаксически допустимый путь к файлу или папке на диске в системе Windows или в сети. Необходимо извлечь из пути расширение имени файла, если оно имеется. Например, необходимо извлечь фрагмент `.ext` из пути `c:\folder\file.ext`.

Решение

```
\.[^.\\/:*?"<>|\r\n]+$
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Для извлечения расширения имени файла можно использовать тот же прием, который применялся для извлечения всего имени файла в рецепте 8.23.

Единственное отличие заключается в том, как обрабатывать точку. Регулярное выражение в рецепте 8.23 не включает никаких точек. Инвертированный символьный класс в выражении из рецепта 8.23 просто обеспечивает совпадение с любыми точками, которые могут встретиться в имени файла.

Расширение имени файла начинается с точки. Поэтому мы добавили `‹\.›` в начало регулярного выражения, чтобы обеспечить совпадение с точкой.

Такие имена файлов, как `Version 2.0.txt`, могут содержать несколько точек. Последняя точка является той, что отделяет расширение от имени файла. Само расширение не может содержать точек. Мы указали на это обстоятельство, поместив точку в символьный класс. Внутри символьных классов точка играет роль обычного литерала, поэтому ее можно не экранировать. Якорь `‹$›` в конце регулярного выражения гарантирует, что с выражением будет совпадать фрагмент `.txt`, а не `.0`.

Если испытуемая строка оканчивается обратным слэшем или именем файла, не содержащим ни одной точки, регулярное выражение вообще не будет обнаруживать совпадение. Когда оно действительно будет обнаруживать соответствие, совпадение будет содержать расширение, включая точку, которая отделяет его от имени файла.

См. также

Рецепт 8.19, если заранее неизвестно, хранит ли испытуемая строка допустимый путь Windows.

8.25. Удаление недопустимых символов из имен файлов

Задача

Необходимо удалить последовательности символов, которые недопустимы в именах файлов Windows. Например, имеется строка с названием документа, которую требуется использовать как имя файла по умолчанию, когда пользователь в первый раз щелкнет на кнопке Save (Сохранить).

Решение

Регулярное выражение

```
[\\/:¨*?<>|]+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст

Оставить пустой строку с замещающим текстом.

Диалекты замещающего текста: .NET, Java, JavaScript, PHP, Perl, Python, Ruby

Обсуждение

В именах файлов в системе Windows не допускается использовать символы \/:¨*?<>|. Эти символы используются для отделения имен устройств и папок, для заключения путей в кавычки, в качестве шаблонных символов и для перенаправления ввода-вывода в командной строке.

Совпадение с этими символами легко можно обеспечить с помощью символьного класса ‹[\\/:¨*?<>|]›. Обратный слэш в символьных классах является метасимволом, его необходимо экранировать другим символом обратного слэша. Все остальные символы всегда интерпретируются в символьных классах как литералы.

Для повышения эффективности мы повторяем символьный класс с помощью квантификатора ‹+›. Поэтому если строка содержит непрерывную последовательность недопустимых символов, то вся эта последовательность будет удалена за один проход. Разница в производительности едва ли будет заметна при работе с короткими строками, такими как имена файлов, но это отличный прием, который следует иметь в виду при работе с большими объемами данных, где наверняка будут встречаться длинные последовательности символов, которые следует удалить.

Так как требуется всего лишь удалить нежелательные символы, в качестве замещающего текста в операции поиска с заменой следует использовать пустую строку.

См. также

Рецепт 3.14, где описывается, как реализовать операцию поиска с заменой с использованием фиксированного замещающего текста в разных языках программирования.

9

Разметка и форматы данных

Обработка разметки и данных в различных форматах с помощью регулярных выражений

Основное внимание в заключительной главе будет уделено решению типичных задач, которые возникают при работе с различными языками разметки и форматами данных: HTML, XHTML, XML, CSV и INI. Хотя предполагается по крайней мере базовое знакомство с этими технологиями, тем не менее далее приводятся краткие описания каждой из них, чтобы можно было освежить свои знания, прежде чем погружаться в эти технологии. Описания концентрируются на основных синтаксических правилах, необходимых для обеспечения корректного поиска структур данных в каждом из форматов. Остальные подробности будут вводиться по мере необходимости, когда мы будем сталкиваться с соответствующими проблемами.

Это не всегда очевидно, но некоторые из этих форматов могут оказаться неожиданно сложными в обработке, по крайней мере, с помощью регулярных выражений. При разработке программ для решения большинства задач, рассматриваемых в этой главе, вместо регулярных выражений лучше использовать специализированные парсеры и функции, особенно если точность имеет большое значение (например, если результаты обработки могут повлиять на безопасность). Однако мы не являемся сторонниками устоявшегося мнения, что разметка в стиле XML никогда не должна обрабатываться с применением регулярных выражений. Регулярные выражения с успехом могут использоваться в разных ситуациях, например при редактировании разметки в текстовом редакторе, для выборки данных из ограниченного набора файлов HTML, исправления ошибок в разметке XML или для обработки файлов в форматах,

напоминающих XML. Да, существуют некоторые проблемы, о которых следует знать, но после прочтения этой главы они не станут для вас неожиданностью.

За дополнительной информацией о применении регулярных выражений для выделения лексем в нестандартных форматах данных обращайтесь к рецепту 3.22.

Основные синтаксические правила форматов, охватываемых в этой главе

Далее приводятся основные синтаксические правила для форматов файлов HTML, XHTML, XML, CSV и INI. Следует отметить, что большая часть сложностей, о которых будет рассказано в этой главе, связана с необходимостью обрабатывать ситуации, возникающие при разнообразных отклонениях от приведенных далее правил.

Язык разметки гипертекста (Hypertext Markup Language, HTML)

Язык HTML используется для описания структуры, семантики и представления миллиардов веб-страниц и других документов. Хотя желание попытаться привлечь регулярные выражения к обработке HTML-документов является совершенно естественным, следует знать заранее, что этот язык плохо подходит для обработки жесткими и точными регулярными выражениями. Это особенно верно в отношении увечной разметки HTML, которая часто встречается в веб-страницах частично благодаря чрезвычайной терпимости веб-браузеров к определенным ошибкам в конструировании разметки HTML. В этой главе мы сосредоточимся на правилах обработки ключевых компонентов корректно оформленного документа HTML: элементов (и атрибутов, содержащихся в них), ссылок на символы, комментариев и объявлений типов документов. В этой книге рассматриваются версии HTML 4.01 (выпущенная в 1999 году) и HTML5 (предварительный вариант этого стандарта был выпущен в середине 2012 года).

Основные строительные блоки разметки HTML называются *элементами*. Элементы записываются с помощью *тегов*, которые окружаются угловыми скобками. Обычно элементы имеют как открывающий тег (например, `<html>`), так и закрывающий (например, `</html>`). Открывающий тег элемента может содержать *атрибуты*, которые будут описаны позже. Между тегами находится *содержимое* элемента, которое может состоять из текста и других элементов или оставаться пустым. Элементы могут вкладываться друг в друга, но они не могут перекрываться (например, `<div><div></div></div>` – это допустимое взаиморасположение элементов, а `<div><span></div></span>` – нет). Для некоторых элементов (таких как `<p>`, который обозначает абзац) закрывающий тег является необязательным. Имеется несколько элементов (включая `<br>`, завершающий строку), которые не имеют содержимого и никогда не используют закрывающие теги. Однако даже пустой элемент может иметь атрибуты. Теги пустых

элементов допускается завершать последовательностью символов />, например <br/>. Имена элементов HTML начинаются с символов A–Z. Во всех допустимых именах элементов используются только буквы и цифры. Имена элементов нечувствительны к регистру символов.

Элементы <script> и <style> заслуживают особого внимания, потому что позволяют встраивать в документы программный код на языке сценариев и таблицы стилей. Эти элементы закрываются ближайшим вхождением закрывающих тегов </style> или </script>, даже если они присутствуют внутри комментария или строки, в пределах описания стиля или программного кода.

Атрибуты определяются внутри открывающего тега элемента, сразу же вслед за его именем, и отделяются одним или несколькими пробельными символами. Большинство атрибутов записываются в виде пар имя-значение. В следующем примере показан элемент <a> (anchor – якорь) с двумя атрибутами и содержимым «Click me!»:

```
<a href="http://www.regexcookbook.com"
   title = 'Regex Cookbook'>Click me!</a>
```

Как видно из этого примера, имя атрибута и значение отделяются друг от друга знаком равенства и необязательным пробельным символом. Значение заключается в апострофы или в кавычки. Чтобы использовать в значениях атрибутов кавычки того же типа, что и объемлющие, необходимо использовать ссылки на символы (описываются ниже). Значения атрибутов можно не заключать в кавычки или апострофы, если они не содержат кавычки, апострофы, тяжелые ударения, знаки «равно», «меньше», «больше» или пробельные символы (в виде регулярного выражения это требование выглядит так: ‹^[^"'`=<>\s]+$›). Некоторые атрибуты (такие как selected и checked, используемые некоторыми элементами форм) оказывают воздействие на элемент, содержащий их, уже своим присутствием и не требуют указывать значение. В таких случаях знак равенства, отделяющий имя атрибута от значения, также опускается. Как вариант, в качестве значений таких «минимизированных» атрибутов допускается использовать их имена (например, selected="selected"). Имена атрибутов начинаются с символов A–Z. Во всех допустимых именах атрибутов используются только буквы, дефисы и двоеточия. Атрибуты могут следовать в любом порядке, а их имена нечувствительны к регистру символов.

Стандарт HTML5 определяет более 2000 *мнемонических ссылок на символы*[1] и более миллиона числовых ссылок на символы (все вместе мы будем называть их *ссылками на символы*). Числовые ссылки на символы определяют соответствующие им символы с помощью

---

[1] Для многих символов определено более одного мнемонического имени в HTML5. Например, символ ≈ имеет шесть имен: ≈, ≈, ≈, ≈, ≈ и ≈.

кодовых пунктов Юникода и записываются в формате &#*nnnn*; или &#x*hhhh*;, где *nnnn* – это одна или более десятичных цифр в диапазоне 0–9, а *hhhh* – это одна или более шестнадцатеричных цифр в диапазоне 0–9 и A–F (без учета регистра символов). Мнемонические ссылки на символы записываются в формате &*имямнемоники*; (с учетом регистра символов, в отличие от большинства других аспектов языка HTML) и особенно удобны, когда требуется ввести литерал символа, имеющего специальное назначение в некоторых контекстах, например угловые скобки (< и >), кавычки (") и амперсанд (&). Также часто используется мнемоника   (неразрывный пробел, код 0xA0), удобство которой обусловлено тем, что в представлении документа HTML отображаются все вхождения этого символа, даже если они следуют друг за другом. Пробелы, символы табуляции, разрывы строк обычно отображаются как единственный пробельный символ, даже если подряд следует множество таких символов. Амперсанд (&) не может использоваться иначе как в ссылках на символы.

Комментарии в разметке HTML оформляются следующим образом:

```
<!-- это комментарий -->
<!-- это тоже комментарий, но он располагается
     на нескольких строках -->
```

Содержимое комментария не имеет специального значения и не отображается средствами просмотра. Для совместимости с ранними (до 1995 года) браузерами некоторые разработчики окружают комментариями HTML содержимое элементов <script> и <style>. Современные браузеры игнорируют такие комментарии и обрабатывают сценарии или таблицы стилей обычным образом.

Документы HTML часто начинаются с *объявления типа документа* (неофициально *DOCTYPE*), которое содержит спецификацию допустимого и недопустимого содержимого документа. Объявление DOCTYPE по своему виду напоминает элемент HTML, как видно в следующей строке, используемой для документов, строго следующих определению HTML 4.01 Strict:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html14/strict.dtd">
```

Ниже приводится объявление DOCTYPE в HTML5:

```
<!DOCTYPE html>
```

Наконец, в HTML5 допускается вставлять в документ разделы *CDATA*, но только внутри встроенного содержимого MathML и SVG. Поддержка разделов CDATA была заимствована из XML и используется для экранирования блоков текста. Такие блоки начинаются с последовательности символов <![CDATA[и заканчиваются первым вхождением последовательности]]>.

Такова, в двух словах, физическая структура документа HTML.[1] Помните, что в действительности разметка HTML часто изобилует отклонениями от этих правил, которые успешно распознаются большинством браузеров. Кроме того, следует отметить, что каждый элемент накладывает свои ограничения на содержимое и внутренние атрибуты, при использовании которых документ HTML рассматривается как допустимый. Рассмотрение подобных правил, предъявляемых к содержимому, выходит далеко за рамки этой книги, но издательство O'Reilly выпустило книгу Чака Муссиано (Chuck Musciano) и Билла Кеннеди (Bill Kennedy) «HTML & XHTML: The Definitive Guide»[2], которая является отличным источником этой информации.

Так как по своей структуре разметка HTML очень похожа на разметку XHTML и XML (описываются ниже), многие регулярные выражения в этой главе написаны так, чтобы обеспечить поддержку всех трех языков разметки.

Расширяемый язык разметки гипертекста (Extensible Hypertext Markup Language, XHTML)

Язык XHTML разрабатывался как дальнейшее развитие HTML 4.01 с целью ухода от связи HTML с SGML и перехода на основы XML. Однако развитие HTML продолжается независимо. В настоящее время XHTML5 развивается как часть спецификации HTML5 и будет представлять собой XML-совместимую версию HTML5 без введения новых особенностей. В этой книге охватываются версии XHTML 1.0, 1.1 и 5.[3] Несмотря на то что синтаксис XHTML во многом сохраняет обратную совместимость с HTML, тем не менее существует несколько важных отличий от структуры HTML, описанной выше:

- Документы XHTML могут начинаться с *объявления XML*, такого как `<?xml version="1.0" encoding="UTF-8"?>`.

---

[1] Стандарт HTML 4.01 определяет поддержку некоторых необычных особенностей SGML, включая инструкции обработки (синтаксис которых отличается от синтаксиса XML) и сокращенные элементы разметки, но не рекомендует пользоваться ими. В этой главе мы будем действовать так, как если бы этой поддержки не существовало вовсе, потому что браузеры поступают точно так же и не реализуют эту поддержку. При желании вы можете познакомиться с этими синтаксическими особенностями в приложении B спецификации HTML 4.01, в разделах B.3.5–7. Стандарт HTML5 явно ликвидировал поддержку этих особенностей, которые все равно не используются браузерами.

[2] Чак Муссиано, Билл Кеннеди «HTML и XHTML. Подробное руководство», 6-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

[3] Что касается пропущенных версий: спецификация XHTML 2.0 разрабатывалась консорциумом W3C в течение нескольких лет, после чего работы над ней были прекращены в пользу HTML5. Версии XHTML 3 и 4 были просто пропущены.

- Непустые элементы должны завершаться закрывающими тегами. Пустые элементы должны иметь закрывающий тег или оканчиваться последовательностью />.
- Имена элементов и атрибутов чувствительны к регистру и должны записываться символами нижнего регистра.
- Благодаря использованию префиксов пространства имен XML имена элементов и атрибутов могут помимо символов, допустимых в именах элементов и атрибутов HTML, содержать двоеточие.
- Не допускается использование значений атрибутов, не заключенных в кавычки. Значения атрибутов должны окружаться кавычками или апострофами.
- Все атрибуты должны иметь соответствующие им значения.

Между HTML и XHTML существует еще ряд различий, которые касаются в основном крайних случаев и обработки ошибок, но в большинстве своем не оказывают влияния на регулярные выражения в этой главе. Дополнительную информацию о различиях между HTML и XHTML можно найти по адресу *http://www.w3.org/TR/xhtml1/#diffs* и *http://wiki.whatwg.org/wiki/HTML_vs._XHTML*.

Синтаксис языка разметки XHTML является подмножеством синтаксиса HTML (и HTML5) и следует правилам языка XML, поэтому многие регулярные выражения в этой главе написаны так, чтобы обеспечить поддержку всех трех языков разметки. Рецепты, в которых упоминается «(X)HTML», в равной степени относятся к HTML и XHTML. Как правило, бывает желательно избежать зависимости от использования соглашений только языка HTML или только XHTML, поскольку они часто смешиваются между собой в документах и веб-браузеры обычно воспринимают такую смесь.

Расширяемый язык разметки (Extensible Markup Language, XML)

Язык XML – это многоцелевой язык, предназначенный, в первую очередь, для обмена структурированными данными. Он используется как основа для создания широкого диапазона языков разметки, включая XHTML, который только что рассматривался. В этой книге охватываются версии XML 1.0 и 1.1. Полное описание особенностей языка XML и его грамматики выходит далеко за рамки этой книги, но применительно к нашему случаю существует всего несколько важных отличий от синтаксиса языка HTML, описанного выше:

- Документы XML могут начинаться с объявления XML, такого как `<?xml version="1.0" encoding="UTF-8"?>`, и содержать другие инструкции по обработке, имеющие похожий формат. Например, инструкция `<?xml-stylesheet type="text/xsl" href="transform.xslt"?>` указывает, что к документу должен применяться файл *transform.xslt* преобразований XSL.

- **Объявление DOCTYPE может включать внутренние объявления, заключенные в квадратные скобки. Например:**

```
<!DOCTYPE example [
  <!ENTITY copy "&#169;">
  <!ENTITY copyright-notice "Copyright &copy; 2012, O'Reilly">
]>
```

- **Непустые элементы должны иметь закрывающий тег. Пустые элементы должны либо иметь закрывающий тег, либо оканчиваться символами />.**
- *Имена* **в языке XML (которые подчиняются правилам, применяемым к именам элементов, атрибутов и мнемонических ссылок) чувствительны к регистру символов и могут использовать символы Юникода. В число допустимых символов входят A–Z, a–z, двоеточие и символ подчеркивания, а после первого символа также допускается использовать 0–9, дефис и точку. Подробнее об этом рассказывается в рецепте 9.4.**
- **Употребление значений атрибутов без кавычек не допускается. Значения атрибутов должны заключаться в апострофы или в кавычки.**
- **Все атрибуты должны сопровождаться значениями.**

Существует множество других правил, которых следует придерживаться, чтобы обеспечить создание правильно оформленных документов XML или, если возникнет необходимость, создать свой собственный парсер XML. Однако только что описанных правил (в дополнение к структуре документов HTML, которая была описана выше) обычно бывает вполне достаточно для реализации простых регулярных выражений.

Так как синтаксис XML очень похож на синтаксис HTML и составляет основу разметки XHTML, многие регулярные выражения в этой главе написаны так, чтобы обеспечить поддержку всех трех языков разметки. Рецепты, в которых упоминается «XML», в равной степени относятся к языкам XML, XHTML и HTML.

Значения, разделенные запятыми (Comma-Separated Values, CSV)

CSV – это старый, но все еще широко применяемый формат файлов, используемый для представления табличных данных. Формат CSV поддерживается большинством приложений электронных таблиц и систем управления базами данных и пользуется большой популярностью при организации обмена данными между приложениями. Несмотря на отсутствие какой-либо официальной спецификации формата CSV, в октябре 2005 года была предпринята попытка дать ему общее определение в документе RFC 4180, а кроме того, этот формат был зарегистрирован организацией IANA как тип MIME «text/csv». До публикации указанного документа RFC стандартом де-факто счи-

тались соглашения, используемые в Microsoft Excel. Так как RFC определяет правила, очень похожие на те, что используются программой Excel, такое положение вещей не представляет большой проблемы. Эта глава охватывает формат CSV, определяемый документом RFC 4180 и используемый приложением Microsoft Excel 2003 и более поздних версий.

Как следует из названия, файлы в этом формате содержат список значений, известных как *записи полей*, разделенных запятыми. Каждая строка, или *запись*, располагается в одной текстовой строке. За последним полем в записи символ запятой не указывается. За последней записью в файле может следовать разрыв строки. Все записи в файле должны содержать одинаковое число полей.

Значение каждого поля в формате CSV может заключаться в кавычки. Кроме того, поля вообще могут быть пустыми. Любое поле, содержащее запятые, кавычки или разрывы строк, должно заключаться в кавычки. Кавычки внутри поля должны экранироваться другой, предшествующей ей, кавычкой.

Первая запись в файле CSV иногда используется как заголовок, содержащий имена всех столбцов. Это невозможно определить программным способом исходя из содержимого файла CSV, поэтому некоторые приложения предлагают пользователю указать, как следует интерпретировать первую строку.

Документ RFC 4180 определяет, что начальные и конечные пробелы в значениях поля являются частью значения. В некоторых старых версиях Excel эти пробелы игнорируются, но Excel 2003 и более поздние версии следуют этому указанию RFC. Документ RFC не определяет порядок обработки ошибок, связанных с наличием неэкранированных кавычек или с чем-то другим. В редких случаях обработка этой ошибки, выполняемая в Excel, может приводить к непредсказуемым результатам, поэтому очень важно гарантировать экранирование кавычек. Поля, содержащие кавычки, сами должны заключаться в кавычки, а поля в кавычках не должны содержать начальных или конечных пробелов за пределами кавычек.

Ниже приводится пример фрагмента в формате CSV, демонстрирующий многие из правил, которые мы только что рассмотрели. Он содержит две записи с тремя полями в каждой:

```
aaa,b b,"""c"" cc"
1,,"333, three,
still more threes"
```

В табл. 9.1 только что продемонстрированный фрагмент CSV отображается более наглядно.

Несмотря на то что мы описали правила CSV, которые учитываются в рецептах этой главы, существует огромное количество вариаций того, как различные программы читают и записывают файлы CSV.

Многие приложения даже позволяют в файлах с расширением «csv» использовать в качестве разделителей полей не только запятые, но и другие символы. В число других типичных отклонений входит порядок включения запятых (или других разделителей полей), кавычек и разрывов строк в значения полей и порядок интерпретации начальных и завершающих пробелов в полях без кавычек – они могут игнорироваться или интерпретироваться как литералы.

Таблица 9.1. Пример отображения фрагмента в формате CSV

aaa	b b	¨c¨ cc
1	(*пустое поле*)	333, three, still more threes

Файлы инициализации (INI)

Простой формат INI часто используется для оформления файлов с настройками. Этот формат плохо определен, и в результате существует масса вариантов интерпретации этого формата различными программами и системами. Регулярные выражения в этой главе придерживаются наиболее распространенных соглашений по оформлению файлов INI, которые описываются ниже.

Параметры в файле INI – это пары имя-значение, разделяемые знаком равенства, с возможными дополнительными пробелами и символами табуляции. Значения могут заключаться в апострофы или кавычки, что позволяет включать в значения начальные и конечные пробелы и другие специальные символы.

Параметры могут группироваться в *разделы*, которые начинаются с имени раздела, заключенного в квадратные скобки и находящегося в отдельной строке. Раздел простирается, пока не будет встречено объявление следующего раздела или конец файла. Разделы не могут быть вложенными.

Точка с запятой отмечает начало *комментария*, который простирается до конца строки. Комментарии могут присутствовать в строках с параметрами или объявлениями разделов. Содержимое комментариев не имеет никакого значения.

Ниже приводится пример файла INI с вводным комментарием (отмечающим дату последнего изменения файла), двумя разделами («user» и «post») и тремя параметрами («name», «title» и «content»):

```
; last modified 2012-02-14

[user]
name=J. Random Hacker

[post]
title = Regular Expressions Rock!
content = "Let me count the ways..."
```

9.1. Поиск тегов XML

Задача

Требуется обеспечить совпадение с любыми тегами HTML, XHTML или XML, присутствующими в тексте, с целью удалить, изменить, подсчитать их количество или выполнить какие-либо другие операции.

Решение

Самое лучшее решение зависит от нескольких факторов, включая точность, эффективность и терпимость к приемлемым для вас ошибкам в разметке. Определившись с требованиями, предъявляемыми к конкретной ситуации, можно перейти к определению перечня того, что хотелось бы сделать с результатами. Неважно, предполагается ли удалять теги, выполнять поиск внутри них, добавлять или удалять атрибуты или заменять теги альтернативной разметкой, – в любом случае сначала нужно отыскать их.

Имейте в виду, что это будет длинный рецепт, полный тонкостей, исключений и вариаций. Если вы ищете быстрое решение и не желаете затрачивать усилия на поиск решения, наилучшим образом отвечающего вашим потребностям, возможно, вам стоит сразу перейти к разделу «Теги (X)HTML (не строго)» этого рецепта, где предлагается решение, сочетающее в себе приличную устойчивость к ошибкам в разметке и строгость подхода.

На скорую руку

Это решение является самым простым и используется гораздо чаще, чем можно было бы предположить, но оно включено сюда больше для сравнения и объяснения его недостатков. Оно может быть достаточно хорошим, если заранее известно, с какого рода содержимым придется работать, и последствия неправильной обработки не причинят существенных хлопот. Это регулярное выражение совпадает с символом <, а затем просто продолжает поглощать любые символы до первого встреченного символа >:

```
<[^>]*>
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Допускается наличие символа > в значениях атрибутов

Следующее регулярное выражение тоже достаточно простое; оно дает корректные результаты не во всех случаях. Однако оно прекрасно справляется с возложенной на него задачей, если используется только для обработки фрагментов допустимой разметки (X)HTML. Его преимущество перед предыдущим регулярным выражением состоит в том, что оно корректно проходит символы > в значениях атрибутов:

```
<(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже приводится это же регулярное выражение, дополненное комментариями и отступами для большей удобочитаемости:

```
<
(?: [^>"']  # Символ не в кавычках или...
  | "[^"]*" # Значение атрибута в кавычках или...
  | '[^']*' # Значение атрибута в апострофах
)*
>
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Оба эти регулярных выражения работают идентично, поэтому у себя можно использовать любое из них. Те, кто пишут на JavaScript (без применения библиотеки XRegExp), смогут воспользоваться только первым вариантом, потому что в JavaScript отсутствует поддержка режима свободного форматирования.

Теги (X)HTML (не строго)

Помимо поддержки символов > в значениях атрибутов следующее регулярное выражение имитирует поведение браузеров, которые обычно реализуют не очень строгие требования к именам тегов (X)HTML. Это позволяет регулярному выражению обходить содержимое, которое не выглядит как теги, включая комментарии, объявления DOCTYPE и литералы символов < в тексте. С этой целью в регулярное выражение было добавлено два основных изменения. Во-первых, дополнительная обработка, помогающая обнаруживать начало и конец значения атрибута, например, при наличии кавычек внутри значения атрибута, не заключенного в кавычки. Во-вторых, специальная обработка имени тега, требующая, в частности, чтобы имя начиналось с буквы A–Z. Имя тега сохраняется в обратной ссылке 1 на тот случай, если его потребуется извлечь:

```
</?([A-Za-z][^\s>/]*)(?:=\s*(?:"[^"]*"|'[^']*'|[^\s>]+)|[^>])*(?:>|$)
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

То же выражение в режиме свободного форматирования:

```
<
/?                    # Разрешить закрывающие теги
([A-Za-z][^\s>/]*)    # Сохранить имя тега в обратной ссылке 1
(?:                   # Обработка значения атрибута:
  = \s*               #   Сообщает о начале значения атрибута
  (?: "[^"]*"         #   Значение атрибута в кавычках
    | '[^']*'         #   Значение атрибута в апострофах
    | [^\s>]+         #   Значение атрибута без кавычек
```

```
  )
|                   # Значение, не относящееся к атрибуту:
  [^>]              #   Символ за пределами значения атрибута
)*
(?:>|$)             # Конец тега или строки
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Оба эти регулярных выражения работают идентично, хотя второй вариант не может использоваться в диалекте JavaScript (без использования библиотеки XRegExp), так как в нем отсутствует поддержка режима свободного форматирования.

Теги (X)HTML (строго)

Это регулярное выражение более сложное, чем те, что приводились выше, потому что оно фактически следует правилам оформления тегов (X)HTML, описанным во вводном разделе этой главы. Это не всегда бывает желательно, так как браузеры не строго следуют этим правилам. Другими словами, данное регулярное выражение не будет совпадать с содержимым, которое не выглядит как допустимый тег (X)HTML, что в результате может приводить к потере части содержимого, которое браузерами интерпретируется как тег (например, если в разметке используется имя атрибута, включающее недопустимые символы, или если атрибуты включены в закрывающий тег). Здесь одновременно соблюдаются правила оформления тегов HTML и XHTML, так как для них характерно смешивание соглашений. Имя тега сохраняется в обратной ссылке 1 или 2 (в зависимости от типа тега – открывающий или закрывающий) на тот случай, если его потребуется извлечь:

```
<(?:([A-Z][-:A-Z0-9]*)(?:\s+[A-Z][-:A-Z0-9]*(?:\s*=\s*(?:"[^"]*"|↵
'[^']*'|[^"'`=<>\s]+))?)*\s*/?|/([A-Z][-:A-Z0-9]*)\s*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы сделать выражение менее загадочным, ниже приводится это же выражение в режиме свободного форматирования с комментариями:

```
<
(?:                         # Вариант выбора для открывающих тегов:
([A-Z][-:A-Z0-9]*)          #   Сохранить имя открывающего тега в обратной ссылке 1
(?:                         #   Позволить присутствие нуля или более атрибутов,
  \s+                       #   разделенных пробельными символами
  [A-Z][-:A-Z0-9]*          #   Имя атрибута
  (?: \s*=\s*               #   Разделитель имени-значения в атрибуте
    (?: "[^"]*"             #   Значение атрибута в кавычках
      | '[^']*'             #   Значение атрибута в апострофах
      | [^"'`=<>\s]+        #   Значение атрибута без кавычек (HTML)
    )
  )?                        #   Позволить атрибуты без значений (HTML)
)*
```

```
  \s*                   #   Позволить завершающие пробельные символы
  /?                    #   Позволить самозакрывающиеся теги
|                       # Вариант выбора для закрывающих тегов:
  /
  ([A-Z][-:A-Z0-9]*)    #   Сохранить имя закрывающего тега в обратной ссылке 2
  \s*                   #   Позволить завершающие пробельные символы
)
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Теги XML (строго)

Язык XML имеет очень точные спецификации, и это означает, что пользовательские программы должны строго следовать его правилам. Это существенное отличие от HTML и многострадальных браузеров, обычно используемых для его обработки. По этой причине для XML мы включили только «строгую» версию:

```
<(?:([_:A-Z][-.:\w]*)(?:\s+[_:A-Z][-.:\w]*\s*=\s*(?:"[^"]*"|'[^']*'))*\s*↵
/?|/([_:A-Z][-.:\w]*)\s*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

И снова это же регулярное выражение в режиме свободного форматирования с дополнительными комментариями:

```
<
(?:                     # Вариант выбора для открывающих тегов:
  ([_:A-Z][-.:\w]*)     #   Сохранить имя открывающего тега в обратной ссылке 1
  (?:                   #   Позволить присутствие нуля или более атрибутов,
    \s+                 #   разделенных пробельными символами
    [_:A-Z][-.:\w]*     #   Имя атрибута
    \s*=\s*             #   Разделитель имени-значения в атрибуте
    (?: "[^"]*"         #   Значение атрибута в кавычках
      | '[^']*'         #   Значение атрибута в апострофах
    )
  )*
  \s*                   #   Позволить завершающие пробельные символы
  /?                    #   Позволить самозакрывающиеся теги
|                       # Вариант выбора для закрывающих тегов:
  /
  ([_:A-Z][-.:\w]*)     #   Сохранить имя закрывающего тега в обратной ссылке 2
  \s*                   #   Позволить завершающие пробельные символы
)
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Подобно предыдущему решению, предназначенному для поиска тегов (X)HTML, эти регулярные выражения сохраняют имя тега в обратной ссылке 1 или 2 в зависимости от типа совпавшего тега – открывающий или закрывающий. Регулярное выражение, совпадающее с тегами XML, немного короче, чем версия для (X)HTML, потому что здесь не приходится иметь дело с особенностями, характерными только для разметки HTML (минимизированные атрибуты и значения без кавычек). Кроме того, оно допускает использовать в именах элементов и атрибутов более широкий диапазон символов.

Обсуждение

Некоторые предостережения

Несмотря на то что достаточно часто возникает потребность производить поиск XML-подобных тегов с помощью регулярных выражений, для обеспечения надежности требуется сбалансированность и внимательное отношение к данным, с которыми предстоит работать. Из-за этих сложностей некоторые стараются воздерживаться от использования регулярных выражений при работе с различными разновидностями разметки XML или (X)HTML, предпочитая специализированные парсеры и функции. Такой подход необходимо иметь в виду, потому что эти инструменты обычно оптимизированы и часто включают устойчивые механизмы определения или обработки некорректной разметки. В мире браузеров, например, для манипулирования разметкой HTML, как правило, лучше воспользоваться преимуществами древовидной объектной модели документа (Document Object Model, DOM). В других случаях можно было бы воспользоваться парсером SAX или XPath. Однако время от времени возникают ситуации, когда решения на базе регулярных выражений работают просто прекрасно и их использование имеет определенный смысл.

Чтобы стерилизовать разметку HTML, полученную из источника, не вызывающего доверия, и обезопасить себя от специально подготовленной злонамеренной разметки и атак вида «межсайтовый скриптинг» (XSS), в первую очередь следует преобразовать все символы <, > и & в соответствующие им мнемоники (`<`, `>` и `&`), затем вернуть обратно теги, которые, как известно, являются безопасными (при условии, что они не содержат атрибутов или используют только разрешенные атрибуты). Например, чтобы вернуть на место теги `<p>`, `<em>` и `<strong>` без атрибутов после замены символов <, > и & их мнемониками, достаточно выполнить поиск с заменой с использованием регулярного выражения `‹<(/?)(p|em|strong)>›` и заменить совпадения замещающим текстом «`<$1$2>`» (или «`<\1\2>`» в Python и Ruby). Затем в измененной строке можно безопасно выполнить поиск тегов HTML с применением регулярных выражений, описываемых в этом рецепте.

Учтя эту «реплику в сторону», приступим к анализу регулярных выражений, которые приведены в этом рецепте. Первые два решения для большинства случаев оказываются чересчур упрощенными, но XML-подобные языки разметки они обрабатывают одинаково. Последние три следуют более строгим правилам и ориентированы на соответствующие языки разметки. Однако даже в последних решениях обрабатываются теги как HTML, так и XHTML, потому что они часто смешиваются в одном документе, нередко по неосторожности. Например, автор может использовать самозакрывающийся тег `<br />` из XHTML в документе HTML4 или по ошибке использовать символы верхнего регистра в именах элементов в документе XHTML. HTML5 еще больше стирает грань между синтаксисом HTML и XHTML.

На скорую руку

Преимущество этого решения заключается в его простоте, благодаря которой выражение легко запоминается и вводится, а кроме того, быстро работает. Цена этой простоте – некорректная обработка некоторых допустимых и недопустимых конструкций XML и (X)HTML. Если вы работаете с разметкой, которую пишете сами, и знаете, что появление таких конструкций в вашем испытуемом тексте невозможно, или если вас не беспокоят последствия ошибок, такая цена может быть вполне приемлемой. Другой пример, когда это решение может оказаться подходящим, – если вы работаете с текстовым редактором, позволяющим предварительно просмотреть совпадения с регулярным выражением.

Регулярное выражение начинается с литерала ‹<› (совпадающего с началом тега). Далее следует инвертированный символьный класс и максимальный квантификатор звездочка ‹[^>]*›, которые обеспечивают совпадение с нулем или более следующих далее символов, отличных от >. Им соответствуют имя тега, атрибуты и ведущий или замыкающий символ /. Здесь можно было бы использовать минимальный квантификатор (‹[^>]*?›), но это не даст ничего, кроме замедления работы регулярного выражения, потому что в этом случае возникает большее число возвратов (причина описывается в рецепте 2.13). Конец тега в регулярном выражении сопоставляется с литералом ‹>›.

Если вы предпочитаете использовать точку вместо инвертированного символьного класса ‹[^>]›, это вполне возможно. Точка будет прекрасно выполнять свои функции при условии, что вместе с ней будет использоваться минимальная звездочка (‹.*?›) и будет активирован режим «точке соответствуют границы строк» (в диалекте JavaScript вместо точки можно применить конструкцию ‹[\s\S]*?›). Точка с максимальной звездочкой (шаблон ‹<.*>›) изменяет смысл регулярного выражения, вынуждая его некорректно совпадать со всем подряд от первого символа < до самого последнего символа > в испытуемом тексте, даже если для этого регулярному выражению придется попутно поглотить многочисленные теги.

Настало время обратиться к примерам. Регулярное выражение из раздела «На скорую руку» будет совпадать с каждой из следующих строк текста целиком:

- `<div>`
- `</div>`
- `<div class="box">`
- `<div id="pandoras-box" class="box" />`
- `<!-- comment -->`
- `<!DOCTYPE html>`
- `<< < w00t! >`
- `<>`

Следует заметить, что этот шаблон совпадает не только с тегами. Хуже того, он не сможет правильно найти совпадение с целыми тегами `<input>` в испытуемых строках `<input type="button" value=">>">` или `<input type="button" onclick="alert(2 > 1)">`. Вместо этого регулярное выражение обнаруживает совпадения только до первого символа `>`, который появляется в значениях атрибутов. Аналогичные проблемы наблюдаются с комментариями, разделами `CDATA` в разметке XML, объявлениями DOCTYPE, с программным кодом в элементах `<script>` и везде, где встречаются символы `>`.

Если приходится обрабатывать что-то более сложное, чем самая простая разметка, особенно когда испытуемый текст происходит из разных или неизвестных источников, лучше воспользоваться одним из более надежных решений, которые приводятся далее в этом рецепте.

Допускается наличие символа > в значениях атрибутов

Следующее регулярное выражение, подобно только что описанному и созданному на скорую руку, включено, прежде всего, для сравнения с последующими более надежными решениями. Однако оно соблюдает самые основные правила, необходимые для сопоставления с XML-подобными тегами, и поэтому может лучше соответствовать вашим потребностям при обработке фрагментов допустимой разметки, включающей только элементы и текст. Отличие от предыдущего выражения заключается в том, что оно пропускает символы `>`, встречающиеся в значениях атрибутов. Например, оно будет корректно совпадать с целыми тегами `<input>` в предыдущих примерах испытуемого текста: `<input type="button" value=">>">` и `<input type="button" onclick="alert(2 > 1)">`.

Как и предыдущий вариант, это регулярное выражение начинается и заканчивается литералами символов угловых скобок, с которых начинается и заканчивается тег. В середине находится повторяющаяся несохраняющая группа, содержащая три альтернативы, разделенных оператором выбора `‹|›`.

Безопасная(!) оптимизация эффективности

После прочтения этого раздела может сложиться впечатление, что можно заставить регулярное выражение работать быстрее, добавив квантификатор ‹*› или ‹+› после первого инвертированного символьного класса (‹[^>"']›). В позициях в испытуемом тексте, где регулярное выражение будет находить совпадение, это действительно так. Возможность совпадения сразу с несколькими символами за раз позволит механизму регулярных выражений пропускать большое число ненужных шагов на пути к успешному совпадению.

Однако такое изменение может привести к менее очевидным отрицательным последствиям в тех позициях, где механизм регулярных выражений обнаруживает лишь частичное совпадение. Когда регулярное выражение обнаруживает совпадение с открывающим символом <, за которым не следует символ >, наличие которого обеспечило бы успешное завершение попытки сопоставления, возникает проблема «катастрофических возвратов», описанная в рецепте 2.15. Она обусловлена огромным числом способов объединения внутреннего квантификатора с внешним (после несохраняющей группы) для сопоставления с текстом, следующим за символом <, каждый из которых механизм регулярных выражений вынужден будет опробовать, прежде чем убедиться в безуспешности попытки сопоставления. Будьте внимательны!

В диалектах регулярных выражений, поддерживающих захватывающие квантификаторы или атомарную группировку (диалекты JavaScript и Python не поддерживают ни то, ни другое), можно избежать этой проблемы, пользуясь при этом преимуществами высокой производительности сопоставления сразу с несколькими символами не в кавычках. Более того, есть возможность пойти еще дальше и уменьшить число потенциальных возвратов в другом месте регулярного выражения. Если используемый диалект регулярных выражений поддерживает обе возможности, лучше использовать захватывающие квантификаторы (показаны здесь во втором регулярном выражении), потому что они позволяют сохранить регулярное выражение более коротким и удобочитаемым.

С атомарной группировкой:

```
<(?>(?:(?>[^>"']+)|"[^"]*"|'[^']*')*)>
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Ruby

С захватывающими квантификаторами:

```
<(?:[^>"']++|"[^"]*"|'[^']*')*+>
```

Параметры: нет
Диалекты: Java, PCRE, Perl 5.10, Ruby 1.9

Первая альтернатива – инвертированный символьный класс ‹[^>"']›, совпадающий с любым одиночным символом, отличным от правой угловой скобки (которая закрывает тег), кавычки или апострофа (оба типа кавычек указывают на начало значения атрибута). Эта альтернатива отвечает за совпадение с именами тегов и атрибутов, а также с другими символами, находящимися за пределами значений в кавычках. Такой порядок следования альтернатив выбран намеренно для обеспечения более высокой производительности. Механизмы регулярных выражений испытывают варианты в регулярном выражении в направлении слева направо, поэтому первым следует вариант, вероятность совпадения с которым выше, чем вероятность совпадения с альтернативами для значений в кавычках (тем более что сопоставление производится по одному символу за раз).

Далее следуют альтернативы для сопоставления со значениями атрибутов в кавычках и апострофах (‹"[^"]*"› и ‹'[^']*'›). Использование инвертированных символьных классов в них позволяет продолжать сопоставление, невзирая на наличие символов >, разрывов строк и всего остального, что не является закрывающей кавычкой.

Следует отметить, что в этом решении не предусматривается никакой специальной обработки комментариев, что позволяло бы исключать комментарии или обеспечивать корректное совпадение с ними и с другими специальными узлами в документе. Прежде чем применять это регулярное выражение, обязательно следует определить, содержимое какого вида требуется обработать.

Теги (X)HTML (не строго)

Благодаря двум основным изменениям это регулярное выражение намного ближе имитирует соблюдение свободных правил, которые используют веб-браузеры при идентификации тегов (X)HTML в исходных текстах. Это решение хорошо подходит для ситуаций, когда необходимо копировать поведение браузера или алгоритм парсинга HTML5 и не приходится беспокоиться о том, действительно ли соответствуют совпадающие теги всем правилам, определяющим допустимую разметку. Однако следует иметь в виду, что может попасться ужасающе недопустимая разметка HTML, которую данное выражение не сможет обрабатывать способом, похожим на тот, которым это делают браузеры, потому что синтаксический анализ крайних случаев ошибочной разметки в браузерах производится их собственными уникальными способами.

Самое существенное отличие этого регулярного выражения от предыдущего решения заключается в том, что оно требует, чтобы за левой угловой скобкой (<) следовала буква в диапазоне A–Z или a–z, которой может предшествовать символ / (для закрывающих тегов). Это ограничение исключает возможность совпадения со случайными символами < в тексте, а также с комментариями, с объявлениями DOCTYPE, с объявлениями и инструкциями обработки XML, с разделами CDATA и т. д.

Этот прием не защищает от совпадения с чем-то, что выглядит как тег внутри комментариев, в программном коде на языке сценариев, в содержимом элементов <textarea> и т. д. В разделе «Игнорирование необычных разделов (X)HTML и XML» ниже показано, как обойти эту проблему. Но сначала рассмотрим, как работает это регулярное выражение.

Символ ‹<› начинает совпадение с литерала левой угловой скобки. Конструкция ‹/?›, следующая далее, совпадает с необязательным символом слэша в закрывающих тегах. Далее следует сохраняющая группа ‹([A-Za-z][^\s>/]*)›, совпадающая с именем тега и запоминающая его как обратную ссылку 1. Если ссылаться на имя тега не потребуется (например, когда нужно просто удалить все теги), сохраняющие круглые скобки можно удалить (достаточно лишь сохранить шаблон, заключенный в них). Внутри группы имеются два символьных класса. Первый класс, ‹[A-Za-z]›, устанавливает правило первого символа в имени тега. Следующий класс, ‹[^\s>/]›, допускает совпадение почти с любыми символами, которые являются частью имени. Единственное исключение – пробельные символы (‹\s›, которые отделяют имя тега от следующих за ним атрибутов), > (который заканчивает тег) и / (используемый перед завершающим символом > в единичных тегах XHTML). Любые другие символы (даже включая кавычки) интерпретируются как часть имени тега. Это может показаться слишком свободным требованием, но именно так действует большинство браузеров. Ошибочные теги могут не оказывать никакого влияния на способ отображения страницы, но они тем не менее будут доступны в дереве DOM и не будут отображаться как текст, хотя любое содержимое, имеющееся внутри них, будет отображаться.

Вслед за именем тега следует подвыражение, обрабатывающее атрибуты, которое было существенно изменено по сравнению с предыдущим регулярным выражением с целью обеспечить более точную имитацию поведения браузеров при парсинге крайних случаев недопустимой разметки. Так как случайный символ > может завершить тег, если он находится не внутри значения атрибута, важно точно определить, где начинается и где заканчивается значение атрибута. Это не так просто, как могло бы показаться, потому что возможно появление кавычки и знака равно внутри тега, но за пределами определения атрибута и даже внутри значения атрибута, не заключенного в кавычки.

Рассмотрим несколько примеров. Данное регулярное выражение совпадет с каждой из следующих строк полностью:

- <em title=">">
- <em !=">">
- </em// em
- <em title=">"">
- <em title=""em">[1]
- <em" title=">">

---

[1] Значением атрибута title является пустая строка, а не em.

В следующих строках регулярное выражение совпадет только с подчеркнутой частью:

- <em "> ">
- <em="> ">
- <em title=="> ">[1]
- <em title=em="> ">
- <em title= ="> ">

Имейте в виду, что обработка этих примеров специально проектировалась так, чтобы соответствовать типичному поведению браузеров.

Вернемся к обработке атрибутов: далее следует несохраняющая группа ‹(?:=\s*(?:"[^"]*"|'[^']*'|[^\s>]+)|[^>])*›. В ней имеются две внешние альтернативы, разделенные оператором выбора ‹|›.

Первая альтернатива, ‹=\s*(?:"[^"]*"|'[^']*'|[^\s>]+)›, совпадает со значением атрибута; знак равенства сигнализирует о начале значения. Вслед за знаком равенства и необязательным пробельным символом (‹\s*›) следует вложенная несохраняющая группа, включающая три альтернативы: ‹"[^"]*"› совпадает со значениями в кавычках, ‹'[^']*'› совпадает со значениями в апострофах и ‹[^\s>]+› совпадает со значениями, не заключенными в кавычки. Шаблон, соответствующий значениям, не заключенным в кавычки, допускает присутствие любых символов, кроме пробельных и >, – даже кавычки и знаки равенства. Это правило является более свободным, чем определено стандартом для допустимой разметки HTML, но оно полностью соответствует поведению браузеров. Обратите внимание: из-за того что альтернатива, совпадающая со значениями, не заключенными в кавычки, благополучно поглощает кавычки, она должна быть помещена в конец списка альтернатив; в противном случае две другие альтернативы (совпадающие со значениями в кавычках и апострофах) никогда не получат шанс найти совпадение.

Вторая альтернатива во внешней группе – это простой символьный класс ‹[^>]›. Она используется для сопоставления с именами атрибутов (по одному символу за раз), пробельными символами, отделяющими атрибуты, завершающим символом / в самозакрывающихся тегах и любыми другими случайными символами в пределах тегов. Так как альтернатива с этим символьным классом совпадает также со знаком равенства (в дополнение к почти любым другим символам), она должна находиться в конце группы, иначе другая альтернатива, совпадающая со значениями атрибутов, никогда не получат шанс участвовать в совпадении.

Далее регулярное выражение заканчивается конструкцией ‹(?:>|$)›. Она соответствует либо концу тега, либо концу строки, смотря что будет достигнуто раньше.

---

[1] Значением атрибута title является =", а не >. Второй знак равно интерпретируется как первый символ значения атрибута, не заключенного в кавычки.

Об управлении возвратами

Ранее во врезке «Безопасная(!) оптимизация эффективности» показано, как повысить скорость сопоставления с тегами при помощи атомарной группировки и захватывающих квантификаторов. В данном случае производительность потенциально может быть улучшена еще больше, потому что наборы символов, которые могут совпасть с шаблонами ‹[^\s>/]*›, ‹[^\s>]+› и ‹[^>]›, перекрываются, обеспечивая тем самым огромный набор возможных комбинаций, которые механизму регулярных выражений придется опробовать, прежде чем оставить неудачную попытку сопоставления.

Фактически, как уже отмечалось выше, мы решили эту проблему, разрешив считать успешным частичное совпадение с регулярным выражением при встрече конца испытуемой строки. Однако если в используемом диалекте регулярных выражений доступна атомарная группировка или захватывающие квантификаторы, их определенно стоит использовать. На то есть две причины. Во-первых, когда механизм возвратов находится под полным контролем, можно без всякой опаски потребовать, чтобы все совпадения завершались закрывающей угловой скобкой >. Иными словами, подвыражение ‹(?:>|$)› в конце регулярного выражения можно заменить литералом ‹>›, ничуть не беспокоясь о проблеме катастрофических возвратов. Во-вторых, этот прием повысит устойчивость регулярного выражения при внесении в него изменений. В текущей его версии даже незначительные изменения могут стать источником проблем, связанных с неконтролируемыми возвратами, и поэтому должны тщательно обдумываться и тестироваться.

Так давайте же воспользуемся этими средствами управления возвратами! Следующие изменения также могут быть перенесены в регулярные выражения, совпадающие с открывающими/одиночными и закрывающими тегами, которые приводятся чуть выше:

С применением атомарной группировки:

```
</?([A-Za-z](?>[^\s>/]*))(?>=\s*(?:"[^"]*"|'[^']*'|[^\
s>]+)|[^>])*(?:>|$)
```

Параметры: нет

Диалекты: .NET, Java, PCRE, Perl, Ruby

С применением захватывающих квантификаторов:

```
</?([A-Za-z][^\s>/]*+)(?:=\s*(?:"[^"]*"|'[^']*'|[^\
s>]+)|[^>])*+(?:>|$)
```

Параметры: нет

Диалекты: Java, PCRE, Perl 5.10, Ruby 1.9

Диалекты JavaScript и Python не поддерживают ни атомарную группировку, ни захватывающие квантификаторы, однако того же эффекта можно добиться, имитируя атомарную группировку с помощью обратных ссылок на совпадения, сохраненные в опережающих проверках (подробнее об этом рассказывается в разделе «Проверки являются атомарными» в рецепте 2.16).

С применением имитации атомарной группировки:

```
</?([A-Za-z](?=([^\s>/]*))\2)(?=((?:=\s*(?:"[^"]*"|'[^']*'|[^\s>]+)|↵
[^>])*))\3(?:>|$)
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Позволив успешное совпадение по достижении конца строки, даже если конец тега так и не был встречен, мы не только сымитировали поведение большинства браузеров, но и решили потенциальную проблему катастрофических возвратов (рецепт 2.15). Если бы мы вынудили регулярное выражение выполнять возвраты (и в конечном счете признать неудачу сопоставления) при отсутствии символа >, завершающего тег, механизму регулярных выражений пришлось бы опробовать все возможные перестановки, допускаемые перекрывающимися шаблонами и вложенными повторяющимися группами, что могло бы породить проблему падения производительности. Однако данная реализация регулярного выражения решает эту проблему и всегда должна обеспечивать высокую эффективность.

Следующие регулярные выражения демонстрируют, как можно адаптировать этот шаблон для совпадения только с открывающими, только с закрывающими или с одиночными (самозакрывающимися) тегами:

Только открывающие и одиночные теги

```
<([A-Za-z][^\s>/]*)(?:=\s*(?:"[^"]*"|'[^']*'|[^\s>]+)|[^>])*(?:>|$)
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В этой версии отсутствует конструкция ‹/?›, следовавшая за открывающей угловой скобкой ‹<›.

Только закрывающие теги

```
</([A-Za-z][^\s>/]*)(?:=\s*(?:"[^"]*"|'[^']*'|[^\s>]+)|[^>])*(?:>|$)
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В данном случае символ слэша за открывающей угловой скобкой ‹<› сделан обязательной частью совпадения. Следует отметить, что мы умышленно допустили возможность появления атрибутов в закры-

вающих тегах, потому что это регулярное выражение основано на «не строгом» решении. Хотя браузеры не используют атрибуты, содержащиеся в закрывающих тегах, тем не менее они не возражают против их существования.

Теги (X)HTML (строго)

Говоря о строгости этого решения, мы подразумеваем, что оно пытается следовать синтаксическим правилам HTML и XHTML, описанным во вводном разделе этой главы, вместо того чтобы имитировать работу браузеров при парсинге исходного кода разметки документа. По сравнению с предыдущими регулярными выражениями эта строгость добавляет следующие правила:

- Имена тегов и атрибутов должны начинаться с буквы A–Z или a–z, а внутри имен могут использоваться только символы A–Z, a–z, 0–9, дефис и двоеточие. В виде регулярного выражения это выглядит так: `‹^[A-Za-z][-:A-Za-z0-9]*$›`.
- После имени тега недопустимы ошибочные, случайные символы. За именем тега могут следовать только пробельные символы, атрибуты (со значениями или без) и необязательный завершающий символ слэша (/).
- В значениях атрибутов, не заключенных в кавычки, не могут использоваться ", ', `, =, <, > и пробельные символы. В виде регулярного выражения это выглядит так: `‹^[^"'`=<>\s]+$›`.
- Закрывающие теги не могут включать атрибуты.

Так как шаблон разбит на две альтернативы, имя тега сохраняется в обратной ссылке с номером 1 или 2 в зависимости от типа совпавшего тега. Первая альтернатива предназначена для поиска открывающих и одиночных тегов, а вторая – для поиска закрывающих тегов. Оба набора сохраняющих круглых скобок можно убрать, если обратные ссылки на имена тегов не нужны.

В следующих примерах две альтернативы выделены в отдельные регулярные выражения. Обе они сохраняют имя тега в обратной ссылке 1:

Открывающие и одиночные теги

```
<([A-Z][-:A-Z0-9]*)(?:\s+[A-Z][-:A-Z0-9]*(?:\s*=\s*↵
(?:"[^"]*"|'[^']*'|[^"'`=<>\s]+))?)*\s*/?>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Конструкция ‹/?›, стоящая непосредственно перед закрывающей угловой скобкой ‹››, позволяет регулярному выражению совпадать как с открывающими, так и с одиночными тегами. Если ее удалить, выражение будет совпадать только с открывающими тегами. Если удалить только квантификатор знак вопроса (сделав обязательным совпадение с символом ‹/›), оно будет совпадать только с одиночными тегами.

Закрывающие теги

```
</([A-Z][-:A-Z0-9]*)\s*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В последних двух разделах говорилось, как увеличить потенциальную производительность за счет добавления атомарной группировки или захватывающих квантификаторов. Строго определенные пути поиска совпадений с этим регулярным выражением (и с его приведенными вариантами) подразумевают отсутствие потенциальных совпадений с одной и той же строкой более чем одним способом; вследствие этого снижается вероятность возвратов, которые могут порождать проблемы. Это регулярное выражение *не рассчитывает* на возможность возвратов, поэтому, если возникнет желание, можно каждый последний квантификатор ‹*›, ‹+› и ‹?› сделать захватывающим (или получить тот же эффект с помощью атомарной группировки), после чего получившиеся регулярные выражения будут совпадать (и не совпадать) ровно с теми же строками с некоторым снижением вероятности возвратов для всего регулярного выражения. Мы отказываемся от подобных вариантов для данного и последующих выражений, чтобы постараться в этом рецепте сохранить контроль над сумасшедшим числом возможностей.

Ниже, в разделе «Игнорирование необычных разделов (X)HTML и XML», демонстрируется, как избежать совпадения с тегами внутри комментариев, в тегах `<script>` и т. п.

Теги XML (строго)

Язык XML не допускает возможность «не строгого» решения вследствие его точной спецификации и требований к парсерам не обрабатывать неправильно оформленную разметку. Хотя можно использовать предыдущие регулярные выражения при обработке документов XML, их упрощенный подход не обеспечивает более высокую надежность поиска, потому что не существует нестрогих программ для работы с разметкой XML, чье поведение можно было бы имитировать.

Это регулярное выражение в своей основе является упрощенной версией регулярного выражения из раздела «Теги (X)HTML (строго)», потому что появилась возможность убрать из него поддержку двух особенностей HTML, не допустимых в XML: значений атрибутов без кавычек и минимизированных атрибутов (атрибутов, не сопровождаемых значением). Еще одно отличие заключено в наборе символов, которые допускается использовать в именах тегов и атрибутов. Фактически правила определения имен в языке XML (применяемые к именам тегов и атрибутов) более свободны, чем показано здесь, и допускают использование сотен тысяч дополнительных символов Юникода. Если потребуется обеспечить допустимость этих символов при поиске, можно заменить три вхождения конструкции ‹[_:A-Z][-.:\w]*› на один из шаблонов из рецепта 9.4.

Примечательно, что список допустимых символов может изменяться в зависимости от используемой версии XML.

Как и в случае с регулярными выражениями для (X)HTML, имя тега сохраняется в обратной ссылке 1 или 2 в зависимости от типа совпавшего тега – открывающий/одиночный или закрывающий. Здесь также можно удалить сохраняющие круглые скобки, если ссылки на имена тегов не требуются.

В следующих примерах два альтернативных шаблона выделены в отдельные регулярные выражения. В результате каждый из них сохраняет имя тега в обратной ссылке 1:

Только открывающие и одиночные теги

```
<([_:A-Z][-.:\w]*)(?:\s+[_:A-Z][-.:\w]*\s*=\s*↵
(?:"[^"]*"|'[^']*'))*\s*/?>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Конструкция </?>, стоящая непосредственно перед закрывающей угловой скобкой <>>, позволяет регулярному выражению совпадать как с открывающими, так и с одиночными тегами. Если ее удалить, выражение будет совпадать исключительно с открывающими тегами. Если удалить только квантификатор знак вопроса, оно будет совпадать исключительно с одиночными тегами.

Только закрывающие теги

```
</([_:A-Z][-.:\w]*)\s*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В следующем разделе демонстрируется, как избежать совпадения с тегами внутри комментариев, в разделах CDATA и в объявлениях DOCTYPE.

Игнорирование необычных разделов (X)HTML и XML

При попытке сопоставления с XML-подобными тегами внутри исходного файла или в строке наибольшую проблему представляет необходимость избежать совпадения с содержимым, которое выглядит как тег, когда его местоположение или контекст явно свидетельствуют, что это не тег. Регулярные выражения для работы с разметкой (X)HTML и XML, которые демонстрировались в этом рецепте, избегают совпадения с проблематичным содержимым за счет ограничений, накладываемых на первый символ в имени элемента. Некоторые идут еще дальше, требуя, чтобы теги полностью соответствовали синтаксическим правилам (X)HTML или XML. Однако для полной надежности необходимо также избегать совпадения с содержимым, находящимся в комментариях, в программном коде на языке сценариев (где могут использоваться символы «больше» и «меньше» в математических операциях), в разделах CDATA языка XML и в ряде других конструкций. Эту проблему можно

решить, если сначала отыскать эти проблематичные разделы, а затем выполнять поиск тегов только за пределами этих совпадений.

В рецепте 3.18 показано, как реализовать поиск между совпадениями с другим регулярным выражением. В этом рецепте используется два шаблона: внутреннее и внешнее регулярное выражение. Решения, которые были продемонстрированы выше, будут играть роль внутреннего выражения. Внешнее регулярное выражение показано ниже, с отдельными шаблонами для (X)HTML и XML. Этот прием скрывает проблематичные разделы от внутреннего регулярного выражения и тем самым позволяет сохранить реализацию достаточно простой.

Вместо поиска между совпадениями с внешним регулярным выражением может оказаться удобнее просто удалить все такие совпадения (например, заменить все совпадения с внешним регулярным выражением пустыми строками). После этого можно выполнить поиск тегов XML или (X)HTML, не заботясь о необходимости пропускать необычные разделы, такие как блоки CDATA и теги `<script>`, так как они будут удалены из испытуемого текста.

Внешнее регулярное выражение для (X)HTML

Следующее регулярное выражение совпадает с комментариями, разделами CDATA и некоторыми специальными элементами. В состав совпадения со специальными элементами `<script>`, `<style>`, `<title>` и `<xmp>`[1] входит все их содержимое. Регулярное выражение также совпадает с элементом `<plaintext>`[2], при обнаружении которого совпадение простирается до конца испытуемой строки:

```
<!--.*?-->|<!\[CDATA\[.*?]]>|<(script|style|textarea|title|xmp)↵
\b(?:[^>"']|"[^"]*"|'[^']*')*>.*?</\1\s*>|<plaintext↵
\b(?:[^>"']|"[^"]*"|'[^']*')*>.*
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Для тех, кому эта строка покажется неудобочитаемой, ниже приводится это же регулярное выражение в режиме свободного форматирования с дополнительными комментариями:

---

[1] `<xmp>` – это малоизвестный, но широко поддерживаемый элемент, похожий на `<pre>`. Подобно `<pre>`, он сохраняет все пробельные символы и по умолчанию использует моноширинный шрифт, но кроме этого, все его содержимое (включая теги HTML) отображается как простой текст. Элемент `<xmp>` был объявлен устаревшим в HTML 3.2 и был полностью удален в HTML 4.

[2] Элемент `<plaintext>` похож на элемент `<xmp>`, за исключением того, что его действие простирается не до закрывающего тега, а до конца документа. Кроме того, как и элемент `<xmp>`, он объявлен устаревшим в HTML 4.0, но продолжает поддерживаться.

```
# Комментарий
<!-- .*? -->
|
# раздел CDATA
<!\[CDATA\[ .*? ]]>
|
# Специальные элементы и их содержимое
<( script | style | textarea | title | xmp )\b
  (?:[^>"']|"[^"]*"|'[^']*')*
> .*? </\1\s*>
|
# Совпадение с <plaintext/> простирается до конца испытуемой строки
<plaintext\b
  (?:[^>"']|"[^"]*"|'[^']*')*
> .*
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Ни одно из выражений, приведенных выше, не будет работать корректно в JavaScript без использования библиотеки XRegExp, потому что в JavaScript отсутствуют режимы «точке соответствуют границы строк» и «режим свободного форматирования». Следующее регулярное выражение снова возвращено к неудобочитаемому представлению, и в нем символ точки заменила конструкция ‹[\s\S]›, благодаря чему его можно использовать в диалекте JavaScript:

```
<!--[\s\S]*?-->|<!\[CDATA\[[\s\S]*?]]>|<(script|style|textarea|title|xmp)↵
\b(?:[^>"']|"[^"]*"|'[^']*')*>[\s\S]*?</\1\s*>|<plaintext↵
\b(?:[^>"']|"[^"]*"|'[^']*')*>[\s\S]*
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Эти регулярные выражения представляют нечто вроде дилеммы: так как они совпадают с тегами <script>, <style>, <textarea>, <title>, <xmp> и <plaintext>, эти теги никогда не будут обнаруживаться вторым (внутренним) регулярным выражением, хотя оно способно находить все теги. Однако решение этой проблемы – всего лишь вопрос добавления дополнительного программного кода, обрабатывающего эти теги отдельно.

Внешнее регулярное выражение для XML

Это регулярное выражение совпадает с комментариями, с разделами CDATA и с объявлениями DOCTYPE. В каждом из этих случаев совпадение обеспечивается отдельными шаблонами, которые объединяются в общее регулярное выражение с помощью метасимвола выбора ‹|›:

```
<!--.*?--\s*>|<!\[CDATA\[.*?]]>|<!DOCTYPE\s(?:[^<>"']|"[^"]*"|↵
'[^']*'|<!(?:[^>"']|"[^"]*"|'[^']*')*>)*>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Это же выражение в режиме свободного форматирования:

```
# Комментарий
<!-- .*? --\s*>
|
# Раздел CDATA
<!\[CDATA\[ .*? ]]>
|
# Объявление типа документа
<!DOCTYPE\s
  (?: [^<>"'] # Неспециальный символ
    | "[^"]*" # Значение в кавычках
    | '[^']*' # Значение в апострофах
    | <!(?:[^>"']|"[^"]*"|'[^']*')*> # Объявление разметки
  )*
>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

А это версия, работающая в диалекте JavaScript (где отсутствуют режимы «точке соответствуют границы строк» и «режим свободного форматирования»):

```
<!--[\s\S]*?--\s*>|<!\[CDATA\[[\s\S]*?]]>|<!DOCTYPE\s(?:[^<>"']|"[^"]*"|↵
'[^']*'|<!(?:[^>"']|"[^"]*"|'[^']*')*>)*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Только что представленные регулярные выражения допускают присутствие пробельных символов между -- и > в конце комментариев XML за счет использования шаблона ‹\s*›. Этим они отличаются от версии, представленной в разделе «Внешнее регулярное выражение для (X)HTML», потому что HTML5 и веб-браузеры отличаются этой особенностью от XML. Обсуждение различий в оформлении комментариев в XML и HTML можно найти в разделе «Поиск допустимых комментариев HTML» в рецепте 9.9.

См. также

Поиск совпадения с любым или со всеми тегами – распространенная задача, но нередко бывает необходимо обеспечить совпадение с каким-нибудь определенным тегом или с одним из небольшого набора; в рецепте 9.2 показано, как решить обе эти задачи. В рецепте 9.3 описывается, как обеспечить совпадение со всеми тегами, кроме перечисленных в списке.

Рецепт 9.4, где описываются символы, допустимые в именах элементов XML и их атрибутов.

В рецепте 9.7 демонстрируется поиск тегов, содержащих определенный атрибут, а в рецепте 9.8 – поиск тегов, не содержащих определенный атрибут.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.13 описывается, как обеспечить совпадение с как можно меньшим количеством символов. В рецепте 2.14 объясняются особенности атомарной группировки. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

9.2. Заменить тег <b> тегом

Задача

Необходимо заменить в испытуемом тексте все открывающие и закрывающие теги `<b>` соответствующими тегами `<strong>`, сохраняя любые существующие атрибуты.

Решение

Следующее регулярное выражение совпадает с открывающим и закрывающим тегами `<b>` как с атрибутами, так и без них:

```
<(/?)b\b((?:[^>"']|"[^"]*"|'[^']*')*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В режиме свободного форматирования:

```
<
(/?)          # Сохранить необязательный ведущий слэш в обратной ссылке 1
b \b          # Дополнить имя тега границей слова
(             # Сохранить любые атрибуты в обратной ссылке 2
  (?: [^>"']  # Любые символы, кроме >, " или '
    | "[^"]*" # Значение атрибута в кавычках
    | '[^']*' # Значение атрибута в апострофах
  )*
)
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Чтобы сохранить все атрибуты при изменении имени тега, можно использовать следующий замещающий текст:

```
<$1strong$2>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
<\1strong\2>
```

Диалекты замещающего текста: Python, Ruby

Чтобы удалить все атрибуты, выполняя ту же процедуру, достаточно опустить обратную ссылку 2 в замещающем тексте:

```
<$1strong>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
<\1strong>
```

Диалекты замещающего текста: Python, Ruby

Программный код, необходимый для реализации этой задачи, приводится в рецепте 3.15.

Обсуждение

Предыдущий рецепт (9.1) включает подробное обсуждение множества способов сопоставления с *любыми* XML-подобными тегами. Это освобождает данный рецепт от необходимости рассматривать прямолинейный подход к проблеме поиска конкретного тега. Тег `<b>` и его замена `<strong>` предложены только в качестве примера, но вы можете подставить на их место имена двух любых других тегов.

Регулярное выражение начинается сопоставлением с литералом `‹<›` – с первым символом любого тега. Далее следует сопоставление с необязательным символом слэша, который обнаруживается в закрывающих тегах, с помощью конструкции `‹/?›`, заключенной в сохраняющие круглые скобки. Сохранение результата сопоставления с этим шаблоном (который будет либо пустой строкой, либо символом слэша) позволит легко восстановить слэш в замещающем тексте без использования какой-либо условной логики.

Затем производится сопоставление самого имени тега, `<b>`. При желании здесь можно использовать имя любого другого тега. Мы задействовали режим нечувствительности к регистру символов, чтобы также обеспечить совпадение с символом верхнего регистра `B`.

Про границу слова (`‹\b›`), следующую за именем тега, легко забыть, но это один из наиболее важных элементов регулярного выражения. Граница слова позволяет обеспечить совпадение только с тегами `<b>`, но не с `<br>`, `<body>`, `<blockquote>` или любыми другими тегами, которые начи-

наются с буквы «b». Для решения этой же проблемы после имени можно было бы использовать сопоставление с пробельным символом (‹\s›), но в этом случае выражение не будет совпадать с тегами, в которых отсутствуют атрибуты и которые поэтому не имеют пробельных символов после имени. Граница слова позволяет решить эту задачу просто и элегантно.

При работе с XML и XHTML не забывайте, что в именах тегов XML допускается использовать двоеточие для отделения пространства имен, а также дефис и некоторые другие символы, которые образуют границу слова. Например, регулярное выражение может обнаружить совпадение с чем-нибудь, похожим на <b-sharp>. Если в вашем случае это имеет значение, вместо границы слова можно воспользоваться опережающей проверкой ‹(?=[\s/>])›. Она позволит обеспечить невозможность совпадения с частью имени тега, причем это решение более надежное.

Шаблон ‹((?:[^>"']|"[^"]*"|'[^']*')*)›, следующий за именем тега, используется для совпадения со всем остальным внутри тега, вплоть до закрывающей угловой скобки. Если обернуть этот шаблон в сохраняющую группировку, как было сделано в этом примере, можно будет легко вставлять обратно любые атрибуты и другие символы (такие как завершающий символ слэша в одиночных тегах) в замещающем тексте. Внутри сохраняющих круглых скобок шаблон повторяет сопоставление с несохраняющей группой, содержащей три варианта выбора. Первый, ‹[^>"']›, совпадает с любым одиночным символом, кроме >, " или '. Другие два варианта совпадают с целой строкой, заключенной в кавычки или апострофы, что позволяет обеспечить совпадение с любыми значениями атрибутов, содержащими правые угловые скобки, которые не смогут интерпретироваться регулярным выражением как конец тега.

Варианты

Замена списков тегов

Если необходимо отыскать любой тег из списка, потребуется внести простое изменение. Нужно все желаемые имена тегов заключить в группу и разделить их оператором выбора. Группа в данном случае используется для ограничения области действия операторов выбора (‹|›).

Следующее регулярное выражение совпадает с открывающими и закрывающими тегами <b>, <i>, и <big>. Замещающий текст, следующий еще ниже, замещает все эти теги соответствующими тегами или , при этом сохраняются все атрибуты:

```
<(/?)([bi]|em|big)\b((?:[^>"']|"[^"]*"|'[^']*')*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже приводится то же выражение в режиме свободного форматирования:

```
<
(/?)              # Сохранить необязательный ведущий слэш в обратной ссылке 1
([bi]|em|big) \b # Сохранить имя тега в обратной ссылке 2
(                 # Сохранить атрибуты и пр. в обратной ссылке 3
  (?: [^>"']      # Любой символ, кроме >, " или '
    | "[^"]*"     # Значение атрибута в кавычках
    | '[^']*'     # Значение атрибута в апострофах
  )*
)
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Для совпадения с тегами <b> и <i> мы использовали символьный класс ‹[bi]›, вместо указания их имен по отдельности через символ оператора выбора ‹|›, как это сделано для тегов и <big>. Символьный класс работает быстрее, чем оператор выбора, потому что для выполнения своей работы ему не требуется выполнять возвраты. Когда искомые варианты выбора отличаются единственным символом, лучше использовать символьный класс.

Кроме того, для имени тега была добавлена сохраняющая группа, в результате чего группа, сопоставляемая с атрибутом, теперь сохраняет свое совпадение в обратной ссылке 3. Несмотря на то что в операции замены всех совпадений тегом нет никакой необходимости ссылаться на имя тега, тем не менее наличие сохраненного имени тега в отдельной обратной ссылке поможет при необходимости определять тип совпавшего тега.

Чтобы сохранить все атрибуты при замене имени тега, можно использовать следующие примеры замещающего текста:

```
<$1strong$3>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
<\1strong\3>
```

Диалекты замещающего текста: Python, Ruby

Чтобы удалить атрибуты совпавшего тега, достаточно убрать обратную ссылку 3 из замещающего текста:

```
<$1strong>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
<\1strong>
```

Диалекты замещающего текста: Python, Ruby

См. также

Рецепт 9.1, где показано, как отыскивать любые XML-подобные теги, обеспечивая достаточную терпимость к ошибкам в разметке.

Рецепт 9.3, который является противоположностью по отношению к этому рецепту и показывает, как отыскивать любые теги, за исключением тех, что в списке.

Приемы, использовавшиеся в регулярных выражениях и в замещающем тексте в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках. В рецепте 2.21 демонстрируется, как вставлять текст, совпавший с сохраняющей группой, в текст замены.

9.3. Удаление всех XML-подобных тегов, за исключением и

Задача

Необходимо удалить все теги в испытуемом тексте, за исключением `<em>` и `<strong>`.

В другом случае требуется не только удалить все теги, отличные от `<em>` и `<strong>`, но также удалить теги `<em>` и `<strong>`, содержащие атрибуты.

Решение

Это прекрасный повод задействовать негативную опережающую проверку (описывается в рецепте 2.16). Применительно к этой задаче негативная опережающая проверка позволит обеспечить совпадение с фрагментами текста, похожими на тег, за исключением случаев, когда сразу за открывающей угловой скобкой < или </ следует определенное слово. Если затем все найденные совпадения заменить пустыми строками (как это сделать, показано в рецепте 3.14), в тексте останутся только требуемые теги.

Решение 1: совпадение с тегами, за исключением и

```
</?(?!(?:em|strong)\b)[a-z](?:[^>'"]|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В режиме свободного форматирования:

```
< /?                  # Разрешить совпадение с закрывающими тегами
(?!
  (?: em | strong ) # Список тегов, исключаемых из сопоставления
  \b                  # Граница слова позволит исключить совпадение
                      # с частью слова
)
[a-z]                 # Первый символ в имени тега должен быть a-z
(?: [^>"']            # Любой символ, кроме >, " или '
  | "[^"]*"           # Значение атрибута в кавычках
  | '[^']*'           # Значение атрибута в апострофах
)*
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Решение 2: совпадение с тегами, за исключением и , и с любыми тегами, содержащими атрибуты

Внеся одно изменение (заменив ‹\b› на ‹\s*›), можно заставить это регулярное выражение совпадать также с любыми тегами и , содержащими атрибуты:

```
</?(?!(?:em|strong)\s*>)[a-z](?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

И снова это же регулярное выражение в режиме свободного форматирования:

```
< /?                  # Разрешить совпадение с закрывающими тегами
(?!
  (?: em | strong ) # Список тегов, исключаемых из сопоставления
  \s* >               # Исключить теги, содержащие атрибуты
)
[a-z]                 # Первый символ в имени тега должен быть a-z
(?: [^>"']            # Любой символ, кроме >, " или '
  | "[^"]*"           # Значение атрибута в кавычках
  | '[^']*'           # Значение атрибута в апострофах
)*
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Обсуждение

Регулярные выражения в этом рецепте имеют много общего с выражениями, приводившимися выше в этой главе и обеспечивающими совпа-

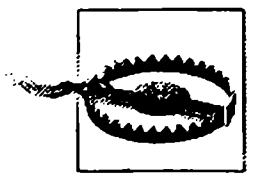
дение с XML-подобными тегами. Кроме негативной опережающей проверки, добавленной с целью предотвратить совпадение с некоторыми тегами, эти регулярные выражения практически эквивалентны выражениям из раздела «Теги (X)HTML (не строго)» в рецепте 9.1. Другое значительное отличие заключается в том, что здесь имя тега не сохраняется в обратной ссылке 1.

Рассмотрим поближе нововведения в этом рецепте. Выражение в решении 1 никогда не будет совпадать с тегами и независимо от того, имеют они атрибуты или нет, но будет совпадать с любыми другими тегами. Выражение в решении 2 будет совпадать с теми же тегами, что и выражение в решении 1, и дополнительно совпадать с тегами и , содержащими один или более атрибутов. В табл. 9.2 приводится несколько примеров испытуемых строк, иллюстрирующих сказанное выше.

Таблица 9.2. Несколько примеров испытуемых строк

Испытуемая строка	Решение 1	Решение 2
	Совпадает	Совпадает
	Совпадает	Совпадает
	Совпадает	Совпадает
	Не совпадает	Не совпадает
	Не совпадает	Не совпадает
	Не совпадает	Совпадает

Поскольку цель применения этих регулярных выражений состоит в том, чтобы заменить совпадения пустыми строками (то есть удалить теги), решение 2 надежнее защищает от возможности изменять форматирование неожиданным образом с помощью допустимых тегов и и творить другие глупости.



В этом рецепте (до сих пор) мы преднамеренно избегали выражения «белый список», описывая, как оставить нетронутыми всего несколько тегов, потому что у этой фразы есть второе значение, связанное с темой безопасности. Существует множество способов обойти ограничения этого шаблона, используя специально подготовленный злонамеренный код разметки HTML. Если вас беспокоит проблема возможных атак с применением злонамеренного кода HTML и атак типа «межсайтовый скриптинг» (XSS), безопаснее всего будет преобразовать все символы <, > и & в соответствующие им мнемонические ссылки (<, > и &) и вставить обратно теги, которые, как известно, безопасны (пока они не содержат атрибуты или используют только атрибуты из списка допустимых атрибутов). style – пример небезопасного атрибута, потому что некоторые браузеры позволяют встраивать в таблицы CSS фрагменты программного кода на

языке сценариев. Например, чтобы после замены <, > и & мнемоническими ссылками вставить теги и обратно, можно выполнить поиск с помощью регулярного выражения `<(/?)(em|strong)>` и заменить совпадения с помощью замещающего текста «<$1$2>» (для диалектов Python и Ruby – «<\1\2>»).

Варианты

«Белый список» для определенных атрибутов

Рассмотрим следующие новые требования: необходимо обеспечить совпадение со всеми тегами, кроме <a>, и , с двумя исключениями. Все теги <a>, имеющие атрибуты, отличные от href или title, должны совпадать; кроме того, должны совпадать теги и , имеющие вообще хоть какие-нибудь атрибуты. Все совпавшие фрагменты должны быть удалены.

Другими словами, следует удалить все теги, кроме перечисленных в «белом списке» (<a>, и). Единственными допустимыми атрибутами являются href и title, причем они допускаются только внутри тегов <a>. В случае появления любого другого атрибута в любом теге весь тег должен удаляться.

Следующее регулярное выражение решает эту задачу:

```
<(?!(?:em|strong|a(?:\s+(?:href|title)\s*=\s*(?:"[^"]*"|'[^']*'))*)\s*>)↵
[a-z](?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

И это же регулярное выражение в режиме свободного форматирования:

```
< /?                          # Разрешить совпадение с закрывающими тегами
(?!
  (?: em                      # Не совпадать с <em>
    | strong                  #   или <strong>
    | a                       #   или <a>
      (?:                     # причем только с теми тегами <a>,
                              # которые не содержат
        \s+                   #   атрибуты, отличные от href и/или title
        (?:href|title)
        \s*=\s*
        (?:"[^"]*"|'[^']*') # Значение атрибута в кавычках
                              #   или апострофах
      )*
  )
  \s* >      # Избегать совпадения с этими тегами, когда они содержат
)            #   только атрибуты, перечисленные выше
[a-z]        # Первый символ в имени тега должен быть буквой a-z
(?: [^>"']   # Любой символ, кроме >, " или '
  | "[^"]*"  # Значение атрибута в кавычках
  | '[^']*'  # Значение атрибута в апострофах
```

```
)*
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Мы сами определили границы области, для которой есть смысл применять такое сложное регулярное выражение. Если правила становятся более сложными, чем это, вероятно, лучше будет написать дополнительный программный код, основываясь на рецептах 3.11 и 3.16, и реализовать в нем проверку каждого совпавшего тега, чтобы определить, как его обрабатывать (исходя из имени тега, содержащихся в нем атрибутов или какой-то иной информации).

См. также

Рецепт 9.1, где показано, как отыскивать любые XML-подобные теги, обеспечивая достаточную терпимость к ошибкам в разметке.

Рецепт 9.2, который является противоположностью по отношению к этому рецепту и показывает, как отыскивать только перечисленные теги.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

9.4. Сопоставление с именами XML

Задача

Требуется проверить, является ли испытуемая строка допустимым *именем* в языке разметки XML (обобщенной синтаксической конструкцией). В языке XML точно определено, какие символы могут составлять имена, и эти правила применяются к именам элементов, атрибутов и мнемоник, к инструкциям обработки и ко многому другому. Первым символом имени может быть буква, символ подчеркивания или двоеточие, остальная часть имени может представлять собой любую комбинацию букв, цифр, символов подчеркивания, двоеточий, дефисов и точек. Это приблизительное описание, но очень близкое к истине. Точный список допустимых символов зависит от используемой версии XML.

Кроме того, может потребоваться добавить шаблон, совпадающий с допустимыми именами, в другие регулярные выражения, предназначенные для работы с XML, когда дополнительная точность влечет за собой дополнительную сложность.

Ниже приводятся некоторые примеры допустимых имен:

- `thing`
- `_thing_2_`
- `:Российские-Вещь`
- `fantastic4:the.thing`
- `日本の物`

Следует отметить, что в число допустимых входят также символы из алфавитов, отличных от Latin, включая даже иероглифы, как видно в последнем примере. Точно так же вслед за первым символом может следовать любая цифра Юникода, а не только арабские цифры 0–9.

Для сравнения ниже приводятся несколько примеров недопустимых имен, которые не должны совпадать с выражением:

- `thing!`
- `thing with spaces`
- `.thing.with.a.dot.in.front`
- `-thingamajig`
- `2nd_thing`

Решение

Подобно идентификаторам во многих языках программирования, в языке XML определено множество символов, которые могут появляться в именах, причем только часть их может употребляться в качестве первого символа. Эти перечни символов существенно отличаются для версии XML 1.0, четвертое издание (и ниже), и версий XML 1.1 и 1.0, пятое издание. По сути, в версии XML 1.1 в именах могут использоваться любые символы, допустимые в четвертом издании версии 1.0, плюс более миллиона других символов. Однако большинство дополнительных символов – это не что иное, как просто позиции в таблице Юникода. Многие из них еще не присвоены символам, но уже считаются допустимыми для совместимости в будущем – по мере расширения базы данных символов Юникода.

Ради лаконичности, когда в этом рецепте мы будем ссылаться на версию XML 1.0, будут подразумеваться издания с первого по четвертое версии XML 1.0. Когда будет говориться об именах XML 1.1, также будет подразумеваться пятое издание версии XML 1.0. Пятое издание стало официальной рекомендацией консорциума W3C только в конце ноября 2008 года, почти через пять лет после появления XML 1.1.

Регулярные выражения в этом рецепте содержат якорные метасимволы, совпадающие с началом и концом текста (‹^···$›), что обеспечивает возможность совпадения с испытуемым текстом только целиком. Если у вас появится необходимость вставлять эти шаблоны

в более длинные регулярные выражения, выполняющие сопоставление с элементами XML, не забудьте удалить якори в начале и в конце шаблона. Якорные метасимволы описываются в рецепте 2.5.

Имена XML 1.0 (приблизительно)

```
^[:_\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nl}][:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, XRegExp, PCRE, Perl, Ruby 1.9

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, чтобы можно было использовать свойства Юникода (‹\p{···}›). В языке PHP поддержка UTF-8 включается с помощью модификатора шаблона /u.

Свойства Юникода не поддерживаются в JavaScript (без использования XRegExp), Python и Ruby 1.8. Регулярное выражение для имен XML 1.1, которое следует ниже, не использует свойства Юникода и поэтому может быть лучшим выбором в этих языках программирования. В разделе «Обсуждение» этого рецепта подробно описывается, почему может быть лучше использовать решение для XML 1.1, даже в диалектах, поддерживающих свойства Юникода.

Имена XML 1.1 (точно)

Ниже следуют две версии одного и того же регулярного выражения, учитывающие различия между диалектами. Во второй версии для определения кодовых пунктов Юникода вместо ‹\u···› используется конструкция ‹\x{···}›.

```
^[:_A-Za-z\u00C0-\u00D6\u00D8-\u00F6\u00F8-\u02FF\u0370-\u037D\u037F-↵
\u1FFF\u200C\u200D\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCF↵
\uFDF0-\uFFFD][:_\-.A-Za-z0-9\u00B7\u00C0-\u00D6\u00D8-\u00F6\u00F8-↵
\u036F\u0370-\u037D\u037F-\u1FFF\u200C\u200D\u203F\u2040\u2070-\u218F↵
\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCF\uFDF0-\uFFFD]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)

Диалекты: .NET, Java, JavaScript, Python, Ruby 1.9

```
^[:_A-Za-z\x{C0}-\x{D6}\x{D8}-\x{F6}\x{F8}-\x{2FF}\x{370}-\x{37D}\x{37F}-↵
\x{1FFF}\x{200C}\x{200D}\x{2070}-\x{218F}\x{2C00}-\x{2FEF}\x{3001}-↵
\x{D7FF}\x{F900}-\x{FDCF}\x{FDF0}-\x{FFFD}][:_\-.A-Za-z0-9\x{B7}\x{C0}-↵
\x{D6}\x{D8}-\x{F6}\x{F8}-\x{36F}\x{370}-\x{37D}\x{37F}-\x{1FFF}\x{200C}↵
\x{200D}\x{203F}\x{2040}\x{2070}-\x{218F}\x{2C00}-\x{2FEF}\x{3001}-↵
\x{D7FF}\x{F900}-\x{FDCF}\x{FDF0}-\x{FFFD}]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)

Диалекты: Java 7, PCRE, Perl

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, чтобы можно было использовать метапоследовательности ‹\x{···}› для значений выше шестнадцатеричного числа FF. В языке PHP поддержка UTF-8 включается с помощью модификатора шаблона /u.

В Ruby 1.8 поддержка Юникода регулярными выражениями отсутствует полностью, однако в разделе «Варианты» этого рецепта приводятся возможные альтернативные решения, обладающие меньшей точностью.

Несмотря на наше утверждение, что эти регулярные выражения точно следуют правилам, применяемым к именам в версии XML 1.1, в действительности это справедливо только для символов с 16-битными кодами (позиции от 0x0 до 0xFFFF), входящими в состав Плоскости 0, или Базовой многоязыковой плоскости (Basic Multilingual Plane). Дополнительно в XML 1.1 допускается использовать после первого символа имени еще 917503 кодовых пунктов, расположенных между позициями 0x10000 и 0xEFFFF (Плоскости 1–14). Только PCRE, Perl, Python и Ruby 1.9 способны ссылаться на кодовые пункты выше 0xFFFF, но вам едва ли доведется встретиться с ними в природе (прежде всего потому, что большая часть позиций в этом диапазоне вообще не присвоена каким-либо символам). Если возникнет необходимость добавить поддержку для этих дополнительных кодовых пунктов, добавьте в конец второго символьного класса:

Java 7, PCRE, Perl

```
‹\x{10000}-\x{EFFFF}›
```

Python

```
‹\U00010000-\U000EFFFF›
```

Ruby 1.9

```
‹\u{10000}-\u{EFFFF}›
```

Но даже без добавления этого массивного диапазона перечень символов имен в XML 1.1 намного обширнее, чем в XML 1.0.

Синтаксис в языке Python, когда за ‹\U› следует восемь шестнадцатеричных цифр, обусловлен его синтаксисом определения литералов строк. Описание наиболее важных деталей, касающихся этой особенности, приводится в рецепте 2.7.

Обсуждение

Многие регулярные выражения в этой главе выполняют сопоставление с элементами XML, поэтому данный рецепт в значительной степени служит целям обеспечения полноты обсуждения шаблонов, которые могут использоваться, когда требуется четко определить, как следует выполнять сопоставление с именами тегов и атрибутов. В остальных рецептах мы стараемся придерживаться более простых и менее точных шаблонов, отдавая предпочтение удобочитаемости и эффективности.

Итак, погрузимся чуть глубже в правила, которые стоят за этими шаблонами.

Имена XML 1.0

Для определения правил, предъявляемых к именам, в спецификации XML 1.0 используется прием «белого списка» и явно перечисляются все допустимые символы. Первым символом имени может быть двоеточие (:), подчеркивание (_) и почти любой символ из следующих категорий Юникода:

- Строчная буква (Ll)
- Заглавная буква (Lu)
- Буква в регистре названия (Lt)
- Буква без регистра (Lo)
- Цифра (Nl)

Вслед за первым символом в дополнение к уже упомянутым символам могут использоваться дефис (-), точка (.) и любой другой символ из следующих категорий:

- Маркер (M), объединяющий следующие подкатегории: маркер, не занимающий места (Mn), маркер, занимающий место (Mc), и объемлющий маркер (Me)
- Модифицированная буква (Lm)
- Десятичная цифра (Nd)

Эти правила приводят к регулярному выражению в разделе «Решение» этого рецепта. Это регулярное выражение еще раз приводится ниже в режиме свободного форматирования:

```
^                                       # Начало текста
[:_\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nl}] # Первый символ имени
[:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]*     # Символы имени (ноль или более)
$                                       # Конец текста
```

Параметры: режим свободного форматирования (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Ruby 1.9

Снова напомним, что библиотека PCRE должна быть скомпилирована с поддержкой UTF-8. В языке PHP поддержка UTF-8 включается с помощью модификатора шаблона /u.

Следует отметить, что во втором символьном классе вместо отдельных подкатегорий букв (Ll, Lu, Lt, Lo и Lm) использована их базовая категория ‹\p{L}›.

Ранее уже отмечалось, что здесь используются приблизительные правила. Это обусловлено двумя причинами. Во-первых, спецификация XML 1.0 (здесь не имеется в виду пятое издание или более поздняя версия) перечисляет ряд исключений из этих правил. Во-вторых, списки

символов в XML 1.0 основаны на стандарте Юникода 2.0, который был выпущен в 1996 году. В последующих версиях стандарта на Юникод была добавлена поддержка новых алфавитов, символы из которых не допускается использовать в именах XML 1.0.

Однако для приведения регулярного выражения к версии Юникода, используемой механизмом регулярных выражений, то есть для ограничения его возможностью сопоставления с символами Юникода 2.0, пришлось бы написать монстроподобный шаблон, заполненный сотнями диапазонов и кодовых пунктов. Если вы действительно захотите создать такого монстра, обращайтесь к спецификации на четвертое издание XML 1.0 (*http://www.w3.org/TR/2006/REC-xml-20060816*), раздел 2.3 «Common Syntactic Constructs» и приложение B «Character Classes».

Ниже приводятся несколько способов сократить представленные выше регулярные выражения в разных диалектах.

Диалекты Perl и PCRE позволяют объединить строчные буквы (Ll), заглавные буквы (Lu) и буквы в регистре названия (Lt) в специальную категорию «буквы, имеющие регистр» (L&). Кроме того, эти диалекты позволяют опускать фигурные скобки в экранированной последовательности ‹\p{···}›, если внутри скобок используется только один символ. Мы задействовали эту особенность в следующем регулярном выражении, использовав конструкцию ‹\pL\pM› вместо ‹\p{L}\p{M}›:

```
^[:_\p{L&}\p{Lo}\p{Nl}][:_\-.\pL\pM\p{Nd}\p{Nl}]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: PCRE, Perl

Диалект .NET поддерживает операцию вычитания символьных классов, которая используется здесь для вычитания подкатегории Lm из категории L вместо того, чтобы перечислять все остальные подкатегории букв:

```
^[:_\p{L}\p{Nl}-[\p{Lm}]][:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалект: .NET

Диалект Java, так же как PCRE и Perl, позволяет опускать фигурные скобки вокруг односимвольных имен категорий Юникода. Кроме того, следующее регулярное выражение использует преимущества более сложной операции вычитания символьных классов (реализованной как пересечение с инвертированным классом) для вычитания подкатегории Lm из L:

```
^[:_\pL\p{Nl}&&[^\p{Lm}]][:_\-.\pL\pM\p{Nd}\p{Nl}]*$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалект: Java

Диалекты JavaScript (без XRegExp), Python и Ruby 1.8 вообще не поддерживают категории Юникода. Диалект Ruby 1.9 не поддерживает эти только что описанные интересные особенности, но поддерживает более переносимую версию этих регулярных выражений, продемонстрированную в разделе «Решение» этого рецепта.

Имена XML 1.1

В спецификации XML 1.0 была допущена ошибка явной привязки к стандарту Юникода 2.0. Более поздние версии Юникода добавили поддержку еще большего количества символов, часть которых принадлежит алфавитам, которые раньше вообще не принимались во внимание (например, чероки, эфиопский и монгольский). Так как XML претендует на положение универсального формата, была предпринята попытка ликвидировать эту проблему в версиях XML 1.1 и в пятом издании XML 1.0. При определении символов, допустимых в именах, в этих более поздних версиях был выполнен переход от стратегии «белого списка» к стратегии «черного списка», чтобы обеспечить поддержку не только тех символов, что были добавлены начиная с версии Юникода 2.0, но и тех, что будут добавлены в будущем.

Эта новая стратегия, когда разрешено все, что явно не запрещено, улучшает будущую совместимость, а также упрощает и сокращает регулярные выражения, точно следующие правилам. Именно поэтому регулярные выражения для работы с именами XML 1.1 помечены как точные, тогда как регулярные выражения для XML 1.0 – как приблизительные.

Варианты

В некоторых рецептах этой главы (например, рецепт 9.1) сегменты выражения, имеющие дело с именами XML, или не используют никакие ограничения, или запрещают дополнительные алфавиты и другие символы, которые в действительности являются допустимыми. Это было сделано, чтобы упростить их изучение. Однако если необходимо разрешить использование других алфавитов, обеспечивая при этом базовый уровень ограничений (и вам не требуется более точная проверка имен из первых регулярных выражений в этом рецепте), можно воспользоваться следующими регулярными выражениями.

Мы оставили якорные метасимволы начала и конца текста в этих регулярных выражениях, потому что они должны использоваться не самостоятельно, а как часть более длинного выражения.

Это первое регулярное выражение просто исключает возможность совпадения с символами, которые могут использоваться в тегах как символы-разделители, а также не допускает возможность совпадения с цифрой в первом символе:

```
[^\d\s¨'/<=>][^\s¨'/<=>]*
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже приводится еще один более короткий способ добиться того же самого. Вместо использования двух отдельных символьных классов в нем применяется негативная опережающая проверка, чтобы отвергнуть совпадение с цифрой в первом символе. Этот запрет применяется только к первому символу, хотя квантификатор ‹+› после символьного класса позволяет регулярному выражению совпадать с неограниченным числом символов:

```
(?!\d)[^\s¨'/<=>]+
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

См. также

Джон Коуэн (John Cowan), один из разработчиков спецификации XML 1.1, объясняет, какие символы недопустимы в именах XML 1.1 и почему, в своем сообщении по адресу *http://recycledknowledge.blogspot.com/2008/02/which-characters-are-excluded-in-xml.html*.

Документ «Background to Changes in XML 1.0, 5th Edition» по адресу *http://www.w3.org/XML/2008/02/xml10_5th_edition_background.html*, где обсуждается обоснование обратного переноса правил, применяемых к именам в XML 1.1, в версию XML 1.0, пятое издание.

Рецепт 9.1, где показано, как отыскивать любые XML-подобные теги, обеспечивая достаточную терпимость к ошибкам в разметке.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.7 рассказывается о сопоставлении с символами Юникода. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

9.5. Преобразование простого текста в HTML добавлением тегов и

Задача

Имеется самый обычный текст, такой как многострочное значение, отправляемое с помощью формы, который необходимо преобразовать во фрагмент разметки HTML для отображения в веб-странице. Абзацы, отделяемые друг от друга двумя разрывами строки, должны быть заключены в теги ··· . Кроме того, разрывы строк следует заменить тегами <br>.

Решение

Эту задачу можно решить, выполнив четыре простых шага. В большинстве языков программирования только на этапе выполнения двух средних шагов можно извлечь выгоду от применения регулярных выражений.

Шаг 1: замена специальных символов HTML мнемоническими ссылками

Поскольку выполняется преобразование простого текста в разметку HTML, на первом этапе необходимо заменить три специальных символа HTML – &, < и > – мнемоническими ссылками (табл. 9.3). В противном случае получившаяся разметка может приводить к неожиданным результатам при отображении в веб-браузере.

Таблица 9.3. Замена специальных символов HTML

Искомый символ	Замещается на
‹&›	«&»
‹<›	«<»
‹>›	«>»

В первую очередь следует заменить символы амперсанда (&), потому что в ходе операции замены в испытуемый текст будут добавлены дополнительные амперсанды, являющиеся частью мнемонических ссылок.

Шаг 2: замена всех разрывов строк тегом

Регулярное выражение:

```
\r\n?|\n
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\R
```

Параметры: нет
Диалекты: PCRE 7, Perl 5.10

Замещающий текст:

```
<br>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP, Python, Ruby

Шаг 3: замена двойных тегов
 на

Регулярное выражение:

```
<br>\s*<br>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Замещающий текст:

```
</p><p>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP, Python, Ruby

Шаг 4: заключить весь текст в теги …

Этот шаг является простой операцией конкатенации строк и не требует применения регулярных выражений.

Пример для JavaScript

Чтобы объединить все четыре шага, создадим функцию JavaScript с именем `htmlFromPlainText()`. Эта функция принимает строку, обрабатывает ее, выполняя только что описанные шаги, и возвращает новую строку с разметкой HTML:

```
function htmlFromPlainText(subject) {
    // шаг 1 (поиск простого текста)
    subject = subject.replace(/&/g, "&amp;").
                      replace(/</g, "&lt;").
                      replace(/>/g, "&gt;");
    // шаг 2
    subject = subject.replace(/\r\n?|\n/g, "<br>");

    // шаг 3
    subject = subject.replace(/<br>\s*<br>/g, "</p><p>");

    // шаг 4
    subject = "<p>" + subject + "</p>";

    return subject;
}

// Протестировать...
htmlFromPlainText("Test.");           // -> "<p>Test.</p>"
htmlFromPlainText("Test.\n");         // -> "<p>Test.<br></p>"
htmlFromPlainText("Test.\n\n");       // -> "<p>Test.</p><p></p>"
htmlFromPlainText("Test1.\nTest2."); // -> "<p>Test1.<br>Test2.</p>"
htmlFromPlainText("Test1.\n\nTest2."); // -> "<p>Test1.</p><p>Test2.</p>"
htmlFromPlainText("< AT&T >");        // -> "<p>&lt; AT&amp;T &gt;</p>"
```

В конец фрагмента программного кода добавлено несколько примеров результатов работы этой функции применительно к различным испытуемым строкам. Тем, кто не знаком с языком JavaScript, следует обратить внимание на модификатор `/g`, добавляемый в конец каждого литерала регулярного выражения, который вынуждает метод `replace()` заместить все совпадения с шаблоном, а не только самое первое. Метапоследовательность `\n`, присутствующая в строке с испытуемым текстом,

в языке JavaScript добавляет символ перевода строки (позиция 0x0A в таблице ASCII) в строковый литерал.

Обсуждение

Шаг 1: замена специальных символов HTML мнемоническими ссылками

Проще всего реализовать этот шаг, выполнив три отдельные операции поиска с заменой (ранее в табл. 9.3 приводился список замещающих мнемоник). Для глобальных операций поиска с заменой в JavaScript всегда используются регулярные выражения, но в других языках программирования можно получить более высокую производительность, выполняя обычную операцию поиска с заменой по простому тексту.

Шаг 2: замена всех разрывов строк тегом

На этом шаге для поиска разрывов строк в соответствии с соглашениями для Windows/MS-DOS (CRLF), UNIX/Linux/OS X (LF) и устаревшей версии Mac OS (CR) используется регулярное выражение `‹\r\n?|\n›`. Пользователи Perl 5.10 и PCRE 7 могут использовать специальный метасимвол `‹\R›` (обратите внимание на верхний регистр символа R) вместо сопоставления с различными последовательностями, обозначающими разрывы строк.

Замена всех разрывов строк тегами `<br>` перед добавлением тегов абзацев на следующем шаге позволяет сохранить простоту всего действия, так как дает возможность добавить пробельный символ между тегами `</p><p>` в последующей операции поиска с заменой. Это поможет сохранить код разметки HTML удобочитаемым, не смешивая все вместе.

Если предпочтительнее использовать одиночные теги в стиле XHTML, то вместо замещающего текста «`<br>`» следует использовать «`<br•/>`». Кроме того, чтобы учесть это изменение, необходимо изменить регулярное выражение, применяемое на третьем шаге.

Шаг 3: замена двойных тегов
 на

Наличие двух разрывов строки, следующих подряд, отмечает конец одного абзаца и начало другого, поэтому замещающий текст для этого шага содержит закрывающий тег `</p>`, за которым следует открывающий тег `<p>`. Если испытуемый текст состоит из единственного абзаца (то есть в нем отсутствуют двойные разрывы строк), подстановка производиться не будет. На шаге 2 было заменено несколько разрывов строк (вместо которых в тексте были оставлены теги `<br>`), поэтому на данном шаге можно было бы выполнить операцию поиска с заменой простого текста. Однако применение регулярного выражения упрощает работу и позволяет игнорировать пробельные символы между разрывами строки. Дополнительные символы пробела все равно не отображаются в документе HTML.

Если создается фрагмент разметки XHTML, и поэтому разрывы строк были замещены текстом «<br•/>» вместо «
», необходимо заменить регулярное выражение, используемое на этом шаге, на ‹<br•/>\s*<br•/>›.

Шаг 4: заключить весь текст в теги …

Шаг 3 добавляет разметку между абзацами. Теперь необходимо добавить тег в самое начало испытуемого текста и закрывающий тег в самый конец. Этот шаг завершает процесс, будь в тексте 1 абзац или 100.

См. также

Рецепт 4.10, включающий дополнительные сведения о метасимволе ‹\R› из диалектов Perl и PCRE и показывающий, как организовать совпадение с дополнительными, необычными разделителями строк, которые поддерживаются метасимволом ‹\R›.

Рецепт 9.6, где рассказывается, как декодировать именованные и числовые мнемоники в стиле XML.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

9.6. Декодирование мнемоник XML

Задача

Требуется преобразовать все мнемоники, определяемые стандартом XML, в соответствующие им символы. Преобразование должно распознавать не только именованные мнемоники (такие как &, < и "), но и числовые (как в десятичной форме записи, например: Σ или Σ, так и в шестнадцатеричной: Σ, Σ или Σ).

Решение

Регулярное выражение

```
&(?:#([0-9]+)|#x([0-9a-fA-F]+)|([0-9a-zA-Z]+));
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Данное регулярное выражение состоит из трех сохраняющих групп. Но только одна из них принимает участие в каждом конкретном сопоставлении и сохраняет совпавший текст. Использование трех групп, как в данном случае, позволяет легко проверить тип совпавшей мнемоники.

Замена совпадений соответствующими символами

Только что представленное регулярное выражение можно применять в комбинации с программным кодом из рецепта 3.16. Примеры кода, что приводятся ниже, демонстрируют порядок выполнения поиска с заменой, когда замещающий текст генерируется во время выполнения.

При создании собственной функции, генерирующей замещающий текст, определить соответствующий замещающий символ можно с помощью анализа обратных ссылок. Если совпадение найдено группой 1, обратная ссылка 1 будет хранить числовую мнемонику в десятичной форме записи, возможно, с ведущими нулями. Если совпадение найдено группой 2, обратная ссылка 2 будет хранить числовую мнемонику в шестнадцатеричной форме записи, возможно, с ведущими нулями. Если совпадение найдено группой 3, обратная ссылка 3 будет хранить именованную мнемонику. Для преобразования можно использовать объект поиска – словарь, хеш или другую структуру данных, наиболее удобную для отображения именованных мнемоник в соответствующие им символы, – и с его помощью быстро определять символ для использования в замещающем тексте.

В следующем разделе на примере программного кода на JavaScript демонстрируется, как все это делается.

Пример на JavaScript

```
// Принимает совпадение ($0) и обратные ссылки;
// возвращает замещающий текст
function callback($0, $1, $2, $3) {
    var charCode;

    // Объект поиска, отображающий имена в десятичные коды
    // Эквивалентные шестнадцатеричные коды указаны в комментариях
    var names = {
        quot: 34, // 0x22
        amp:  38, // 0x26
        apos: 39, // 0x27
        lt:   60, // 0x3C
        gt:   62  // 0x3E
    };

// Десятичная мнемоника
if ($1) {
    charCode = parseInt($1, 10);
// Шестнадцатеричная мнемоника
} else if ($2) {
    charCode = parseInt($2, 16);
// Именованная мнемоника;
// выполняется отображение с помощью объекта поиска
} else if ($3 && ($3 in names)) {
    charCode = names[$3];
// Недопустимая или неизвестная мнемоника
```

```
    } else {
        return $0; // Вернуть само совпадение
    }

    // Вернуть символ
    return String.fromCharCode(charCode);
}

// Заменить все мнемоники символами
subject = subject.replace(
/&(?:#([0-9]+)|#x([0-9a-fA-F]+)|([0-9a-zA-Z]+));/g,
callback);
```

Обсуждение

Регулярное выражение и пример кода, показанные выше в этом рецепте, созданы для декодирования фрагментов текста в стиле XML, а не полных XML-документов. Регулярное выражение может пригодиться для преобразования содержимого в формате XML или (X)HTML в простой текст, но имейте в виду, что в нем отсутствуют какие-либо ограничения на местонахождение именованных или числовых мнемоник в испытуемом тексте. Например, в нем не предусмотрено никакой специальной обработки, которая позволяла бы пропускать блоки CDATA в XML или сценарии в HTML.

Пример на JavaScript преобразует числовые мнемоники в десятичном и шестнадцатеричном представлениях в соответствующие им символы и дополнительно отображает пять именованных мнемоник, определяемых стандартом XML: `"` (“), `&` (&), `'` ('), `<` (<) и `>` (>). Стандарт HTML определяет еще целое множество именованных мнемоник, которые не были упомянуты здесь.[1] Следуя способу, предложенному в примере кода, легко можно добавить столько именованных мнемоник, сколько потребуется.

Пример на JavaScript превратит следующий текст:

```
"&lt; &bogus; dec &#65;&#0065; &amp;lt; hex &#x41;&#x041; &gt;"
```

в строку:

```
"< &bogus; dec AA &lt; hex AA >"
```

JavaScript не поддерживает кодовые пункты Юникода выше U+FFFF, поэтому данный пример (точнее, метод `String.fromCharCode()`, используемый в нем) дает правильный результат только для числовых мнемоник не выше `&#xFFFF;` (в шестнадцатеричном представлении) и `&#65535;` (в десятичном представлении). В большинстве случаев это не является проблемой, потому что символы с кодами выше этого диапазона редко используются на практике. Числовые мнемоники со значениями выше

---

[1] Стандарт HTML 4.01 определяет 252 именованные мнемоники, а стандарт HTML5 – более 2000.

этого диапазона считались недопустимыми в первой редакции стандарта XML 1.0.

Некоторые языки программирования и библиотеки для работы с XML имеют встроенные функции для декодирования мнемоник XML или HTML. Например, в PHP 4.3 и выше имеется функция `html_entity_decode()`. Но иногда бывает полезно написать свой метод, потому что такие функции отличаются набором распознаваемых ими мнемоник. Некоторые функции, например `CGI::unescapeHTML()` в Ruby, распознают даже меньше пяти именованных мнемоник, определяемых стандартом XML.

См. также

Рецепт 9.5, где объясняется, как преобразовать простой текст в формат HTML добавлением тегов `<p>` и `<br>`. На первом шаге в этом процессе выполняется кодирование символов &, < и > в именованные мнемоники.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

9.7. Поиск определенных атрибутов в XML-подобных тегах

Задача

Требуется отыскать внутри файла (X)HTML или XML теги, содержащие определенный атрибут, такой как `id`.

Этот рецепт охватывает несколько вариантов одной и той же задачи. Предположим, что необходимо обеспечить совпадение со всеми типами строк, приведенными ниже, с помощью отдельных регулярных выражений:

- Теги с атрибутом `id`.
- Теги `<div>` с атрибутом `id`.
- Теги с атрибутом `id`, имеющим значение `my-id`.
- Теги, содержащие имя класса `my-class` внутри значения атрибута `class` (имена классов отделяются пробельными символами).

Решение

Теги с атрибутом id (на скорую руку)

Если необходимо быстро выполнить поиск в текстовом редакторе, обеспечивающем возможность предварительного просмотра результатов,

добиться цели поможет следующее (максимально упрощенное) регулярное выражение:

```
<[^>]+\sid\b[^>]*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже то же регулярное выражение разложено в режиме свободного форматирования:

```
<          # Начало тега
[^>]+      # Имя тега, атрибуты и пр.
\s id \b # Имя искомого атрибута как целое слово
[^>]*      # Оставшаяся часть тега, включая значение атрибута id
>          # Конец тега
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Теги с атрибутом id (более надежно)

В отличие от только что рассмотренного регулярного выражения, следующий подход к этой же задаче поддерживает поиск значений атрибутов в кавычках, содержащих литерал >, и не будет выдавать совпадение для тегов, в которых последовательность символов `id` содержится в одном из значений атрибутов:

```
<(?:[^>"']|"[^"]*"|'[^']*')+?\sid\s*=\s*("[^"]*"|'[^']*')↵
(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В режиме свободного форматирования:

```
<
(?: [^>"']              # Имена тега и атрибута и пр.
  | "[^"]*"             #   и значения атрибутов в кавычках
  | '[^']*'
)+?
\s id                   # Имя искомого атрибута как целое слово
\s* = \s*               # Разделитель имени-значения атрибута
( "[^"]*" | '[^']*' ) # Сохранить значение атрибута в обратной ссылке 1
(?: [^>"']              # Остальные символы
  | "[^"]*"             #   и значения атрибутов в кавычках
  | '[^']*'
)*
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Это регулярное выражение сохраняет значение атрибута `id` и окружающие его кавычки в обратной ссылке 1. Это позволит использовать значение в программном коде за пределами регулярного выражения или в строке замещающего текста. Если значение атрибута не потребуется, можно вместо сохраняющей группировки использовать несохраняющую группу или вообще заменить всю последовательность `‹\s*=\s*("[^"]*"|'[^']*')›` метасимволом `‹\b›`. В этом случае совпадение с оставшейся частью тега, в том числе и со значением атрибута `id`, обеспечит остальная часть регулярного выражения.

Теги с атрибутом id

Чтобы отыскать определенный тег, в начало предыдущего регулярного выражения необходимо добавить его имя и внести еще пару незначительных изменений. В следующем примере после открывающей угловой скобки `‹<›` мы добавили `‹div\s›`. Метасимвол `‹\s›` (пробельный символ) гарантирует невозможность совпадения с тегами, чьи имена просто начинаются с трех букв «div». Заранее известно, что за именем тега обязательно будет следовать пробельный символ, потому что у тегов, которые требуется отыскать, имеется по крайней мере один атрибут (`id`). Дополнительно мы заменили последовательность `‹+?\sid›` на `‹*?\bid›`, чтобы выражение работало, когда `id` является первым атрибутом тега и отсутствуют дополнительные отделяющие символы (кроме первого пробела) после имени тега:

```
<div\s(?:[^>"']|"[^"]*"|'[^']*')*?\bid\s*=\s*("[^"]*"|'[^']*')↵
(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

То же выражение в режиме свободного форматирования:

```
<div \s                 # Имя тега и следующий за ним пробельный символ
(?: [^>"']              # Имена тега и атрибута и пр.
  | "[^"]*"             #   и значения атрибутов в кавычках
  | '[^']*'
)*?
\b id                   # Имя искомого атрибута как целое слово
\s* = \s*               # Разделитель имени-значения атрибута
( "[^"]*" | '[^']*' ) # Сохранить значение атрибута в обратной ссылке 1
(?: [^>"']              # Остальные символы
  | "[^"]*"             #   и значения атрибутов в кавычках
  | '[^']*'
)*
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Теги с атрибутом id, имеющим значение «my-id»

На этот раз, по сравнению с регулярным выражением из раздела «Теги с атрибутом id (более надежно)», мы удалили сохраняющие круглые скобки вокруг значения атрибута `id`, потому что оно уже известно заранее. В частности, подвыражение `‹("[^"]*"|'[^']*')›` мы заменили на `‹(?:"my-id"|'my-id')›`:

```
<(?:[^>"']|"[^"]*"|'[^']*')+?\sid\s*=\s*(?:"my-id"|'my-id')↵
(?:[^>"']|"[^"]*"|'[^']*')*>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

И версия в режиме свободного форматирования:

```
< #
(?: [^>"']              # Имена тега и атрибута и пр.
  | "[^"]*"             #   и значения атрибутов в кавычках
  | '[^']*'
)+?
\s id                   # Имя искомого атрибута как целое слово
\s* = \s*               # Разделитель имени-значения атрибута
(?: "my-id"             # Искомое значение атрибута,
  | 'my-id' )           #   окруженное кавычками или апострофами
(?: [^>"']              # Остальные символы
  | "[^"]*"             #   и значения атрибутов в кавычках
  | '[^']*'
)*
>
```

Параметры: нечувствительность к регистру символов, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

В подвыражении `‹(?:"my-id"|'my-id')›` можно было бы избежать повторения «my-id» (немного потеряв в эффективности), использовав подвыражение `‹(["'])my-id\1›`. Применение сохраняющей группировки и обратной ссылки гарантирует, что искомое значение начинается и заканчивается кавычками одного и того же типа.

Теги, содержащие «my-class» в значении атрибута class

Если в предыдущих регулярных выражениях планка оправданности применения регулярного выражения еще не была преодолена, то в данном случае, очевидно, мы достигли границы, за которой неразумно пытаться решать задачу с помощью единственного регулярного выражения. Разбиение процесса на несколько регулярных выражений поможет в достижении поставленной цели, поэтому мы разобьем поиск на три части. Первое регулярное выражение совпадает с тегами, второе отыскивает в них атрибут `class` (и сохраняет его значение в обратной ссылке) и наконец выполняется поиск значения `my-class`.

Поиск тегов:

```
<(?:[^>"']|"[^"]*"|'[^']*')+>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Рецепт 9.1 посвящен вопросам сопоставления с XML-подобными тегами. Там объясняется, как действует только что представленное регулярное выражение, и демонстрируются альтернативные решения, обеспечивающие различную степень сложности и точности.

Затем с помощью программного кода, который приводится в рецепте 3.13, можно выполнить поиск атрибута `class` внутри каждого совпадения, используя следующее регулярное выражение:

```
^(?:[^>"']|"[^"]*"|'[^']*')+?\sclass\s*=\s*("[^"]*"|'[^']*')
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Оно сохраняет все значение атрибута `class` и окружающие его кавычки в обратной ссылке 1. Совпадение со всем, что находится перед атрибутом `class`, обеспечивает подвыражение `<^(?:[^>"']|"[^"]*"|'[^']*')+?>`, которое за один шаг перешагивает через значение в кавычках, чтобы избежать попыток поиска слова «class» внутри значений других атрибутов. Правая часть выражения заканчивает совпадение, как только будет достигнут конец значения атрибута `class`. Все, что в теге следует дальше, для нас не имеет никакого значения, поэтому нет никаких причин продолжать сопоставление до конца найденного тега.

Символ крышки в начале регулярного выражения привязывает его к началу испытуемого текста. Это никак не влияет на совпадение, но он препятствует механизму регулярных выражений начинать новые попытки сопоставления (которые неизбежно потерпят неудачу) во всех позициях символов в строке, если сопоставление с начала строки оказалось неудачным.

Наконец, если оба предыдущих регулярных выражения благополучно обнаружили совпадение, можно выполнить поиск в значении обратной ссылки 1 второго регулярного выражения, используя следующий шаблон:

```
["'\s]my-class["'\s]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Так как имена классов отделяются пробельными символами, имя `my-class` должно ограничиваться с обеих сторон либо пробельным символом, либо вообще ничем. Если бы не тот факт, что имена классов могут включать дефисы, вместо двух несохраняющих групп можно было бы использовать метасимволы границы слова. Однако дефис создает гра-

ницы слова и поэтому подвыражение `<\bmy-class\b>` может совпасть с частью имени `not-my-class`.

Обсуждение

В разделе «Решение» этого рецепта уже подробно описано, как действуют эти регулярные выражения, поэтому мы не будет повторяться. Не следует забывать, что регулярные выражения часто бывают неидеальным решением задачи поиска в разметке, особенно когда достигается уровень сложности, как в этом рецепте. Прежде чем задействовать такие регулярные выражения, всегда следует посмотреть, не лучше ли использовать альтернативные решения, такие как применение XPath, парсера SAX или использование DOM. Мы включили эти регулярные выражения, потому что найдется немало людей, которые захотят идти подобным путем, но не говорите, что мы вас не предупреждали. Хотелось бы надеяться, что мы смогли показать некоторые из проблем, возникающих при поиске в разметке, и помогли вам избежать еще более непродуманных решений.

Регулярные выражения в этом рецепте написаны исходя из предположения, что значения атрибутов всегда будут заключены в кавычки или апострофы. Значения атрибутов, не заключенные в кавычки, не поддерживаются.

См. также

Рецепт 9.8, решающий концептуально противоположную задачу и отыскивающий теги, не содержащие определенный атрибут.

Рецепт 9.1, где показано, как отыскивать любые XML-подобные теги, обеспечивая достаточную терпимость к ошибкам в разметке.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

9.8. Добавление атрибута cellspacing в теги <table>, где этот атрибут отсутствует

Задача

Необходимо отыскать в файле (X)HTML и добавить `cellspacing="0"` во все таблицы, не имеющие атрибута `cellspacing`.

Этот рецепт служит примером добавления атрибутов в XML-подобные теги, которые еще не имеют этих атрибутов. Вы можете заменить имя тега и атрибута, а также значение атрибута на любое другое.

Решение

Выражение 1: упрощенное решение

Для совпадения с тегами `<table>`, не содержащими слово `cellspacing`, можно использовать негативную опережающую проверку, как показано ниже:

```
<table\b(?![^>]*?\scellspacing\b)([^>]*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

То же выражение в режиме свободного форматирования:

```
<table \b           # Совпадение с "<table",
                    #   за которым следует граница слова
(?!                 # Проверить несовпадение с выражением ниже
  [^>]*?            #   Любые атрибуты и пр.
  \s cellspacing \b #   Совпадение с "cellspacing" как с полным словом
)
([^>]*)             # Сохранить атрибуты и пр. в обратной ссылке 1
>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Выражение 2: более надежное решение

В следующем регулярном выражении мы заменили оба экземпляра инвертированного символьного класса ‹[^>]› из упрощенного решения на ‹(?:[^>"']|"[^"]*"|'[^']*')›. Этот более длинный шаблон поглощает значения атрибутов в кавычках или апострофах за один шаг.

```
<table\b(?!(?:[^>"']|"[^"]*"|'[^']*')*?\scellspacing\b)↵
((?:[^>"']|"[^"]*"|'[^']*')*)>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

А ниже это же выражение в режиме свободного форматирования:

```
<table \b     # Совпадение с "<table" как с полным словом
(?!           # Проверить несовпадение с выражением ниже
  (?:[^>"']|"[^"]*"|'[^']*')*?
  \s cellspacing \b
)
(             # Сохранить атрибуты и пр. в обратной ссылке 1
  (?:[^>"']|"[^"]*"|'[^']*')*
)
>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Вставка нового атрибута

Все регулярные выражения, показанные в решениях 1 и 2, могут использовать один и тот же замещающий текст, потому что все регулярные выражения сохраняют атрибуты совпавшего тега `<table>` (если таковые имеются) в обратной ссылке 1. Это позволяет вставлять атрибуты обратно как часть замещающего значения при добавлении нового атрибута `cellspacing`. Ниже приводятся необходимые замещающие строки:

```
<table•cellspacing="0"$1>
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
<table•cellspacing="0"\1>
```

Диалекты замещающего текста: Python, Ruby

В рецепте 3.15 показано, как реализовать поиск с заменой, когда в замещающем тексте используется обратная ссылка.

Обсуждение

Чтобы понять, как действуют эти регулярные выражения, разберем сначала упрощенное решение. Это решение, как будет показано, содержит четыре логические части.

Первая часть, `‹<table\b›`, совпадает со строкой литералов `<table`, за которой следует граница слова (`‹\b›`). Граница слова предотвращает возможность совпадения с тегами, имена которых просто начинаются с последовательности символов «table». Несмотря на то что это может показаться необязательным при работе с (X)HTML (поскольку в этом языке разметки отсутствуют теги с именами, например, «tablet», «tableau» или «tablespoon»), тем не менее это хорошая практика, которая поможет избежать появления ошибок, когда потребуется адаптировать это регулярное выражение для поиска других тегов.

Вторая часть выражения, `‹(?![^>]*?\scellspacing\b)›`, — это негативная опережающая проверка. Она не поглощает символы испытуемого текста, совпавшие с ней, она проверяет, потерпит ли неудачу попытка сопоставления, если слово `cellspacing` появится где-нибудь в пределах открывающего тега. Так как предполагается добавить атрибут `cellspacing` во все совпадения, необходимо исключить совпадение с тегами, уже имеющими этот атрибут.

Так как опережающая проверка заглядывает за текущую позицию попытки сопоставления, в ней используется начальная конструкция `‹[^>]*?›`, позволяющая в процессе поиска заглядывать вперед настолько, насколько потребуется, пока не будет обнаружено нечто, похожее на конец тега (первое вхождение символа >). Остальная часть подвыражения

в опережающей проверке (‹\scellspacing\b›) просто совпадает со строкой литералов «cellspacing» как с полным словом. Мы предусмотрели проверку совпадения с начальным пробельным символом (‹\s›), потому что он всегда должен отделять имя атрибута от имени тега или от предыдущих атрибутов. Мы предусмотрели проверку совпадения с границей слова вместо совпадения с другим пробельным символом, потому что граница слова полностью обеспечивает совпадение со строкой `cellspacing` как с целым словом, даже если атрибут не имеет значения или если непосредственно за именем атрибута следует знак равенства.

При таком решении, если регулярное выражение найдет строку `cellspacing` перед >, попытка сопоставления потерпит неудачу. Если опережающая проверка не найдет `cellspacing` до того, как ей встретится >, попытка сопоставления сможет продолжиться.

Двигаясь дальше, мы переходим к третьей части регулярного выражения: ‹([^>]*)›. Это инвертированный символьный класс со следующим за ним квантификатором «ноль или более раз», обернутый в сохраняющую группу. Сохранение этой части совпадения позволяет легко возвращать назад атрибуты, присутствующие в совпавшем теге, как часть замещающего текста. В отличие от негативной опережающей проверки, эта часть фактически добавляет атрибуты тега в строку совпадения с регулярным выражением.

Наконец, регулярное выражение совпадает с литералом ‹>› в конце тега.

Регулярное выражение 2, так называемая более надежная версия, действует точно так же, как и только что описанное выражение, за исключением того, что оба экземпляра инвертированного символьного класса ‹[^>]› были заменены на ‹(?:[^>"']|"[^"]*"|'[^']*')›. Это позволило повысить надежность регулярного выражения: во-первых, добавилась поддержка значений атрибутов в кавычках, содержащих символ >, а во-вторых, гарантируется невозможность совпадения с тегами, содержащими в значениях атрибутов слова «cellspacing».

Что касается замещающего текста, он может использоваться в обеих версиях регулярного выражения и будет замещать каждый совпавший тег `<table>` новым тегом, включающим `cellspacing="0"` в качестве первого атрибута, за которым следуют все остальные атрибуты, имевшиеся в оригинальном теге (обратная ссылка 1).

См. также

Рецепт 9.7, решающий концептуально противоположную задачу и отыскивающий теги, содержащие определенный атрибут.

Рецепт 9.1, где показано, как отыскивать любые XML-подобные теги, обеспечивая достаточную терпимость к ошибкам в разметке.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описы-

вается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

9.9. Удаление XML-подобных комментариев

Задача

Требуется удалить комментарии из документа (X)HTML или XML. Например, требуется удалить комментарии из веб-страницы, прежде чем передать ее браузеру, чтобы уменьшить размер файла страницы или чтобы выполнить поиск без риска найти совпадение внутри комментариев.

Решение

Отыскать комментарии совсем несложно благодаря наличию минимальных квантификаторов. Ниже приводится выражение, выполняющее эту работу:

```
<!--.*?-->
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Достаточно прямолинейно. Как обычно, из-за отсутствия в JavaScript режима «точке соответствуют границы строк» необходимо заменить точку символьным классом, включающим все символы, чтобы регулярное выражение могло совпадать с многострочными комментариями. Ниже приводится версия выражения для JavaScript:

```
<!--[\s\S]*?-->
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Чтобы удалить комментарии, нужно заменить все совпадения пустыми строками (то есть ничем). В рецепте 3.14 приводится программный код, выполняющий замену всех совпадений с регулярным выражением.

Обсуждение

Принцип действия

В начале и в конце этого регулярного выражения присутствуют строки литералов ‹<!--› и ‹-->›. Поскольку ни один из этих символов не имеет специального значения в регулярных выражениях (за исключением дефиса, который в символьных классах образует диапазоны), их не требуется экранировать. Остается только исследовать фрагмент ‹.*?› или ‹[\s\S]*?› в середине выражения.

Благодаря режиму «точке соответствуют границы строк» точка в первом регулярном выражении совпадает с любым одиночным символом. В версии для JavaScript место точки занимает символьный класс `[\s\S]`. Однако эти два регулярных выражения полностью эквивалентны. Метасимволу `\s` соответствуют любые пробельные символы, а метасимволу `\S` соответствуют все остальные символы. Их объединение обеспечивает совпадение с любым символом.

Минимальный квантификатор `*?` повторяет предшествующий ему элемент «любой символ» ноль или более раз, минимально возможное число раз. Благодаря этому предшествующая конструкция будет повторяться, пока не будет встречено первое вхождение `-->`, вместо того чтобы сразу совпасть до конца испытуемого текста, а затем выполнять возвраты, пока не будет встречено последнее вхождение `-->`. (Подробнее о том, как выполняются возвраты с минимальными и максимальными квантификаторами, рассказывается в рецепте 2.13.) Эта стратегия прекрасно работает, потому что XML-подобные комментарии не могут вкладываться друг в друга. Другими словами, комментарии всегда оканчиваются самым первым (левым) вхождением `-->`.

Когда нельзя удалять комментарии

Большинству веб-разработчиков известно, что комментарии HTML внутри элементов `<script>` и `<style>` используются для обеспечения обратной совместимости с древними версиями браузеров. В настоящее время это стало бессмысленным шаманством, но оно продолжает жить, отчасти благодаря простому копированию фрагментов разметки при создании новых страниц. Предположим, что при удалении комментариев из документа (X)HTML вы не хотели бы уничтожить программный код JavaScript и таблицы CSS. Вероятно, вам также захочется оставить содержимое элементов `<textarea>`, разделов CDATA и значения атрибутов внутри тегов.

Выше говорилось, что удалить комментарии совсем несложно. Как оказывается, эти слова справедливы, только если игнорировать некоторые необычные области разметки (X)HTML или XML, где изменяются синтаксические правила. Другими словами, если игнорировать сложные части проблемы, то все просто.

Конечно, в некоторых случаях можно было бы оценить разметку, которую требуется обработать, и решить, что эти проблемы вполне можно игнорировать, возможно, потому, что вы сами писали эту разметку и знаете, чего в ней можно ожидать. Упрощенный подход может использоваться при выполнении операции поиска с заменой в текстовом редакторе, позволяющем проверять каждое совпадение перед его удалением.

Но вернемся назад и посмотрим, как обойти эти проблемы. В разделе «Игнорирование необычных разделов (X)HTML и XML» в рецепте 9.1 некоторые из этих проблем уже рассматривались в контексте сопоставления с XML-подобными тегами. Подобный подход можно использо-

вать для поиска комментариев. Чтобы сначала отыскать необычные разделы, можно использовать программный код из рецепта 3.18 и регулярное выражение, которое приводится ниже, а затем замещать комментарии, найденные между совпадениями, пустыми строками (то есть удалить комментарии):

```
<(script|style|textarea|title|xmp)\b(?:[^>"']|"[^"]*"|'[^']*')*>↵
.*?</\1\s*>|<plaintext\b(?:[^>"']|"[^"]*"|'[^']*')*>.*|↵
<[a-z](?:[^>"']|"[^"]*"|'[^']*')*>|<!\[CDATA\[.*?]]>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Добавим несколько пробелов и комментариев в режиме свободного форматирования, чтобы сделать это регулярное выражение более простым для понимания:

```
# Специальный элемент: тег и содержимое
<( script | style | textarea | title | xmp )\b
  (?:[^>"']|"[^"]*"|'[^']*')*
> .*? </\1\s*>
|
# <plaintext/> простирается до конца испытуемой строки
<plaintext\b
  (?:[^>"']|"[^"]*"|'[^']*')*
> .*
|
# Обычный элемент: только тег
<[a-z] # Первый символ имени тега
  (?:[^>"']|"[^"]*"|'[^']*')*
>
|
# Раздел CDATA
<!\[CDATA\[ .*? ]]>
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Ниже приводится эквивалентная версия для диалекта JavaScript, в котором отсутствуют оба режима: «точке соответствуют границы строк» и «режим свободного форматирования»:

```
<(script|style|textarea|title|xmp)\b(?:[^>"']|"[^"]*"|'[^']*')*>↵
[\s\S]*?</\1\s*>|<plaintext\b(?:[^>"']|"[^"]*"|'[^']*')*>[\s\S]*|↵
<[a-z](?:[^>"']|"[^"]*"|'[^']*')*>|<!\[CDATA\[[\s\S]*?]]>
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Варианты

Поиск допустимых комментариев XML

В действительности существует еще несколько синтаксических правил, касающихся комментариев в разметке (X)HTML и XML, помимо того, что комментарии начинаются с `<!--` и заканчиваются `-->`. В частности:

- Комментарии не могут содержать два дефиса, следующие подряд. Например, `<!-- com--ment -->` — это недопустимый комментарий, потому что в нем имеются два дефиса, следующие подряд.
- Закрывающей последовательности не может предшествовать дефис, являющийся частью комментария. Например, `<!-- comment --->` — это недопустимый комментарий, но полностью пустой комментарий `<!---->` допускается.
- Между `--` и `>` в закрывающей последовательности может присутствовать пробельный символ. Например, `<!-- comment -- >` — это полностью допустимый комментарий.

Эти правила легко оформить в виде регулярного выражения:

```
<!--[^-]*(?:-[^-]+)*--\s*>
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обратите внимание, что совпадение с содержимым комментария между открывающей и закрывающей последовательностями является необязательным, поэтому данное выражение будет совпадать с пустым комментарием `<!---->`. Однако если между ограничителями появится дефис, за ним должен следовать хотя бы один символ, отличный от дефиса. А так как внутри комментария не может присутствовать два дефиса, идущих подряд, минимальный квантификатор в регулярном выражении в начале рецепта был заменен максимальными квантификаторами. Минимальные квантификаторы тоже будут работать, но их использование здесь приведет к ненужным возвратам (рецепт 2.13).

Некоторые читатели, посмотрев на это новое регулярное выражение, могут спросить, почему здесь дважды используется инвертированный символьный класс `‹[^-]›` вместо того, чтобы просто поместить дефис в несохраняющую необязательную группу (то есть `‹<!--(?:-?[^-]+)*--\s*>›`). На то есть отличная причина, которая возвращает нас к обсуждению «катастрофических возвратов» из рецепта 2.15.

Так называемые *вложенные квантификаторы* всегда требуют особого внимания, чтобы избежать ситуации катастрофических возвратов. Квантификатор считается вложенным, когда он присутствует внутри группы, которая сама повторяется квантификатором. Например, шаблон `‹(?:-?[^-]+)*›` содержит два вложенных квантификатора: знак вопроса, следующий за дефисом, и знак плюс, следующий за инвертированным символьным классом.

Однако в действительности вложение квантификаторов само по себе не несет угрозы производительности. Опасность, скорее, заключается в потенциально огромном числе способов, которыми внешний квантификатор `(*)` может комбинироваться с внутренними квантификаторами в процессе сопоставления с текстом. Если механизму регулярных выражений не удается отыскать --> в конце частичного совпадения (как это требуется, если включить данный сегмент шаблона в регулярное выражение, совпадающее с комментариями), он должен опробовать все возможные комбинации повторений, прежде чем убедиться в безуспешности попытки и перейти к следующей позиции. Число возможных вариантов увеличивается чрезвычайно быстро с каждым дополнительным символом, который механизм должен попробовать сопоставить. Однако во вложенных квантификаторах нет ничего опасного, если подобная ситуация невозможна. Например, шаблон `(?:-[^-]+)*` не представляет угрозы, несмотря на то что содержит вложенный квантификатор `(+)`, потому что теперь сопоставление с дефисом должно производиться точно один раз на каждом повторении группы и потенциальное число точек возврата увеличивается линейно с увеличением длины испытуемого текста.

Еще один способ избежать проблемы возвратов, описанной только что, заключается в использовании атомарной группировки. Следующее выражение эквивалентно первому в этом разделе, но оно на несколько символов короче и не поддерживается диалектами JavaScript и Python:

```
<!--(?>-?[^-]+)*-->\s*>
```

Параметры: нет

Диалекты: .NET, Java, PCRE, Perl, Ruby

Подробное описание атомарной группировки (и близких к ней захватывающих квантификаторов) приводится в рецепте 2.14.

Поиск допустимых комментариев HTML

Официально в стандарте HTML 4.01 используются правила оформления комментариев, заимствованные из XML и описанные выше, но веб-браузеры никогда не обращали внимания на мелочи. Правила оформления комментариев в HTML5 имеют два отличия от правил, используемых в XML, которые приближают их к правилам, которые фактически реализуются браузерами. Во-первых, пробельные символы между -- и > в закрывающей последовательности считаются недопустимыми. Во-вторых, текст внутри комментария не может начинаться с > или -> (в веб-браузерах, стремящихся завершить комментарии как можно раньше).

Ниже показаны правила оформления комментариев в HTML5, оформленные в виде регулярного выражения:

```
<!--(?!->)[^-]*(?:-[^-]+)*-->
```

Параметры: нет

Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В отличие от предыдущего регулярного выражения, совпадающего с допустимыми комментариями XML, в этом выражении отсутствует ‹\s*› перед завершающим символом ‹>› и добавлена негативная опережающая проверка ‹(?!-?>)› сразу после открывающей последовательности ‹<!--›.

В действительности, веб-браузеры используют более свободные правила оформления комментариев, чем определены стандартом HTML. Именно поэтому обычно предпочтительнее использовать более простые регулярные выражения ‹<!--.*?-->› (с включенным режимом «точке соответствуют границы строк») или ‹<!--[\s\S]*?-->›, представленные в разделе «Решение».

См. также

Рецепт 9.10, где показано, как отыскивать определенные слова, встречающиеся внутри XML-подобных комментариев.

Рецепты 7.5, 7.6 и 7.7, объясняющие, как организовать сопоставление с комментариями в исходном программном коде на различных языках программирования.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.6 говорится о границах слов. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.10 обсуждаются обратные ссылки. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.13 описывается, как обеспечить совпадение с как можно меньшим количеством символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

9.10. Поиск слов в XML-подобных комментариях

Задача

Требуется отыскать все вхождения слова TODO (что сделать) внутри комментариев (X)HTML или XML. Например, требуется отыскать подчеркнутый текст в следующей строке:

```
This "TODO" is not within a comment, but the next one is. ↵
<!-- TODO: Come up with a cooler comment for this example. -->
```

Решение

Существует по меньшей мере два подхода к решению этой проблемы, и каждый из них имеет свои преимущества. Первый, который мы называем «двухступенчатый подход», состоит в том, чтобы отыскать ком-

ментарии с помощью внешнего регулярного выражения, а затем внутри каждого совпадения выполнить поиск с помощью другого регулярного выражения или даже с применением инструментов простого поиска текста. Этот подход дает лучшие результаты, если программный код, выполняющий эту работу, пишете вы сами, потому что деление решения задачи на два этапа упростит реализацию и сделает ее быстрой. Однако если поиск в файлах выполняется с помощью текстового редактора или инструмента grep, подход с делением задачи на два этапа работать не будет, если только используемый инструмент не предлагает специальную возможность поиска внутри совпадений с другим регулярным выражением.[1]

Если необходимо выполнить поиск слов внутри комментариев с применением единственного регулярного выражения, эту задачу можно решить посредством проверки соседних символов. Этот подход демонстрируется в разделе «Одноступенчатый подход».

Двухступенчатый подход

Когда это возможно, лучше разбить решение этой задачи на два этапа: найти комментарии и затем найти в них слово TODO.

Так можно найти комментарии:

```
<!--.*?-->
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Стандартный диалект JavaScript не поддерживает режим «точке соответствуют границы строк», но в нем вместо точки можно использовать символьный класс, соответствующий любому символу, как показано ниже:

```
<!--[\s\S]*?-->
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Затем в каждом из комментариев, найденном с помощью только что представленных регулярных выражений, можно поискать текст, совпадающий с литералами ‹TODO›. При желании это регулярное выражение можно использовать в режиме нечувствительности к регистру символов и с границами слова по обеим сторонам, чтобы обеспечить совпадение только с полным словом TODO, например:

```
\bTODO\b
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

---

[1] PowerGREP, который описывается в разделе «Инструменты для работы с регулярными выражениями» в главе 1, является одним из таких инструментов, способных выполнять поиск внутри совпадений.

В рецепте 3.13 показано, как реализовать поиск в совпадениях с внешним регулярным выражением.

Одноступенчатый подход

Опережающая проверка (описывается в рецепте 2.16) позволяет решить эту задачу с помощью единственного регулярного выражения, хотя и менее эффективно. В следующем регулярном выражении используется позитивная опережающая проверка, чтобы убедиться, что за словом TODO находится последовательность -->, закрывающая комментарий. Сама она не может сказать, находится ли слово внутри комментария или просто за ним следует комментарий, поэтому используется вложенная негативная опережающая проверка, чтобы убедиться в отсутствии открывающей последовательности <!-- перед -->:

```
\bTODO\b(?=(?:(?!<!--).)*?-->)
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Так как стандартный диалект JavaScript не поддерживает режим «точке соответствуют границы строк», вместо точки можно использовать ‹[\s\S]›:

```
\bTODO\b(?=(?:(?!<!--)[\s\S])*?-->)
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Двухступенчатый подход

В рецепте 3.13 приводится программный код, реализующий поиск внутри совпадений с другим регулярным выражением. Он принимает внутреннее и внешнее регулярное выражение. Регулярное выражение, совпадающее с комментарием, используется как внешнее выражение, а ‹\bTODO\b› – как внутреннее. Главное, что необходимо отметить, – это минимальный квантификатор ‹*?›, который следует за символом точки или за символьным классом в регулярном выражении, совпадающем с комментариями. Как описывается в рецепте 2.13, этот квантификатор обеспечивает совпадение до первого вхождения последовательности --> (завершающей комментарий), а не до самого последнего.

Одноступенчатый подход

Это решение более сложное и медленное. Его преимуществом является объединение двух этапов из предыдущего подхода в одно регулярное выражение. Благодаря этому данное решение может использоваться в текстовых редакторах, в средах разработки или с другими инструмента-

ми, не позволяющими производить поиск внутри совпадений с другим регулярным выражением.

Разобьем регулярное выражение на составляющие в режиме свободного форматирования и поближе рассмотрим каждую его часть:

```
\b TODO \b     # Совпадение с символами "TODO" как с полным словом
(?=            # За которыми следуют:
  (?:          #   Группировка несохраняющая:
    (?! <!-- ) #     Отсутствует предшествующая последовательность "<!--"
    .          #     Совпадение с любым единственным символом
  )*?          #   Ноль или более раз, минимально возможное число раз
  -->          #   Совпадение с символами "-->"
)
```

Параметры: точке соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Эта прокомментированная версия регулярного выражения не будет работать в JavaScript без использования библиотеки XRegExp из-за отсутствия в этом диалекте обоих режимов, «свободного форматирования» и «точке соответствуют границы строк».

Следует отметить, что регулярное выражение содержит негативную опережающую проверку, вложенную внутрь позитивной опережающей проверки. Это позволяет потребовать, чтобы любое совпадение с `TODO` предшествовало последовательности `-->` и между ними отсутствовало совпадение с `<!--`.

Если вам понятно, как все это работает, замечательно – вы можете пропустить оставшуюся часть данного раздела. Но если что-то кажется вам непонятным, давайте вернемся немного назад и шаг за шагом построим внешнюю позитивную опережающую проверку в этом регулярном выражении.

Допустим на минуту, что нам требуется просто отыскать вхождения слова `TODO`, за которыми где-то в строке находится последовательность символов `-->`. В результате мы получим выражение `‹\bTODO\b(?=.*?-->)›` (в режиме «точке соответствуют границы строк»), которое совпадет с подчеркнутым фрагментом в строке `<!--TODO-->`. Конструкция `‹.*?›` в начале опережающей проверки совершенно необходима, потому что в противном случае регулярное выражение будет совпадать, только когда за словом `TODO` сразу же будет следовать `-->`, без каких-либо символов между ними. Квантификатор `‹*?›` повторяет точку ноль или более раз, причем минимально возможное число раз, что просто замечательно, так как нам требуется совпадение только до первого вхождения `-->`.

К настоящему моменту регулярное выражение может быть записано так: `‹\bTODO(?=.*?-->)\b›`, где второй метасимвол `‹\b›` перенесен за опережающую проверку без какого-либо влияния на текст совпадения. Это обусловлено тем, что и граница слова, и опережающая проверка являются проверками с нулевой длиной совпадения (раздел «Проверка сосед-

них символов» в рецепте 2.16). Однако для большей удобочитаемости и производительности лучше поставить границу слова первой. В середине частичного совпадения механизм регулярных выражений быстрее проверит наличие границы слова, потерпит неудачу и двинется дальше, пытаясь начать сопоставление со следующего символа в строке, не тратя время на опережающую проверку.

Итак, похоже, что регулярное выражение `‹\bTODO\b(?=.*?-->)›` работает замечательно, но что если применить его к испытуемой строке `TODO <!- - separate comment -->`? Выражение точно так же обнаружит совпадение со словом TODO, потому что за ним следуют символы `-->`, невзирая на то, что слово находится за пределами комментария. То есть необходимо заменить символ точки внутри опережающей проверки, совпадающий с любым символом, на конструкцию, совпадающую с любым символом, не являющимся частью последовательности `<!--`, которая открывает новый комментарий. Мы не можем использовать инвертированный символьный класс, такой как `‹[^<!-]›`, потому что требуется разрешить вхождение символов `<`, `!` и `-`, если они не объединены в точную последовательность `<!--`.

Это как раз та ситуация, когда на помощь может прийти негативная опережающая проверка. Конструкции `‹(?!<!--).›` соответствует любой одиночный символ, не являющийся частью последовательности, открывающей комментарий. Поместив этот шаблон внутрь несохраняющей группировки, `‹(?:(?!<!--).)›`, можно повторять всю последовательность с помощью минимального квантификатора `‹*?›`, который ранее применялся к точке.

Объединив все вместе, получаем окончательное регулярное выражение, которое было приведено как решение этой задачи: `‹\bTODO\b(?=(?:(?!<!--).)*?-->)›`. В JavaScript, где отсутствует режим «точке соответствуют границы строк», эквивалентное выражение имеет вид: `‹\bTODO\b(?=(?:(?!<!--)[\s\S])*?-->)›`.

Варианты

Хотя регулярное выражение из раздела «Одноступенчатый подход» и проверяет, не следует ли за словом `TODO` строка `-->` без `<!--` между ними, но оно не проверяет обратное условие: предшествует ли искомому слову строка `<!--` без `-->` между ними. Есть несколько причин, почему мы пропустили это правило:

- Обычно можно отказаться от выполнения двойной проверки, тем более, что одноступенчатое регулярное выражение предназначено для использования в текстовых редакторах и с другими инструментами, где можно визуально проверить результаты поиска.
- Чем меньше количество проверок, тем меньше на их выполнение тратится времени (то есть достигается выигрыш за счет отказа от дополнительных проверок).

- Самое главное: так как заранее неизвестно, как далеко перед словом TODO находится начало комментария, для заглядывания назад может потребоваться использовать бесконечную ретроспективную проверку, которая поддерживается только диалектом .NET.

При использовании диалекта .NET и при наличии желания включить эту дополнительную проверку можно задействовать следующее регулярное выражение:

```
(?<=<!--(?:(?!-->).)*?)\bTODO\b(?=(?:(?!<!--).)*?-->)
```

Параметры: нечувствительность к регистру символов, точке соответствуют границы строк
Диалект: .NET

Это более строгое регулярное выражение, поддерживаемое только диалектом .NET, содержит в самом начале дополнительную позитивную ретроспективную проверку, которая действует точно так же, как опережающая проверка в конце, но в обратном порядке. Так как ретроспективная проверка заглядывает назад, отыскивая <!--, она содержит вложенную опережающую проверку, которая позволяет ей совпадать с любыми символами, не являющимися частью последовательности -->.

Так как опережающая и ретроспективная проверки в начале выражения являются проверками с нулевой длиной совпадения, окончательное совпадение будет содержать только слово TODO. Совпадения с проверками не включаются в текст окончательного совпадения.

См. также

Рецепт 9.9, где подробно обсуждается, как обеспечить совпадение с комментариями XML.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.4 говорится, что точке соответствуют любые символы. В рецепте 2.6 говорится о границах слов. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.13 описывается, как обеспечить совпадение с как можно меньшим количеством символов. В рецепте 2.16 рассказывается об опережающих и ретроспективных проверках.

9.11. Изменение разделителя, используемого в файлах CSV

Задача

Требуется заменить все запятые, разделяющие поля в файле CSV, на символы табуляции. Запятые, присутствующие внутри полей, окруженных кавычками, должны остаться без изменений.

Решение

Следующее ниже регулярное выражение совпадает с отдельным полем CSV и с предшествующим ему символом-разделителем, если таковой имеется. Обычно предшествующий разделитель является запятой, но он может быть и пустой строкой (то есть ничем) при совпадении с первым полем в первой записи или разрывом строки при совпадении с первым полем любой последующей записи. Каждый раз когда обнаруживается совпадение, само поле, включая возможные окружающие его кавычки, сохраняется в обратной ссылке 2, а предшествующий ему разделитель – в обратной ссылке 1.

Регулярные выражения в этом рецепте предназначены для работы только с допустимыми файлами CSV с точки зрения правил, которые приводились в разделе «Значения, разделенные запятыми (Comma-Separated Values, CSV)» в начале этой главы.

```
(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ниже приводится это же регулярное выражение в режиме свободного форматирования:

```
( , | \r?\n | ^ )  # Сохраняющая группа 1 совпадает с разделителями полей
                   # или с началом строки
(                  # Сохраняющая группа 2 совпадает с единственным полем:
  [^",\r\n]+       #   не заключенным в кавычки
|                  # Или:
  " (?:[^"]|"")* " #   заключенным в кавычки (может содержать
                   #   экранированные кавычки)
)?                 # Группа является необязательной, так как
                   # поле может быть пустым
```

Параметры: режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Используя это регулярное выражение и программный код из рецепта 3.11, можно реализовать итерации по содержимому файла CSV и проверять значение обратной ссылки 1 после каждого совпадения. Значение строки с замещающим текстом для каждого совпадения зависит от значения этой обратной ссылки. Если это запятая, она замещается символом табуляции. Если обратная ссылка содержит пустую строку или разрыв строки, это значение остается на месте (то есть не делается ничего или полученное значение вставляется обратно как часть строки замещающего текста). Так как поля CSV сохраняются в обратной ссылке 2 как часть общего совпадения, их также необходимо вставлять в каждую строку замещающего текста. Единственное, что действительно необходимо замещать, – это запятые, сохраняемые в обратной ссылке 1.

Пример на JavaScript

Следующий ниже фрагмент представляет законченную веб-страницу, включающую два многострочных поля ввода и кнопку с надписью Replace (Заменить) между ними. После щелчка на кнопке из первого текстового поля (помеченного как Input (Вход)) извлекается введенная в него строка, а затем с помощью только что продемонстрированного регулярного выражения все запятые-разделители преобразуются в символы табуляции и получившаяся строка помещается во второе текстовое поле (помеченное как Output (Выход)). Если в первое поле было введено допустимое содержимое в формате CSV, оно должно появиться во втором текстовом поле с запятыми, замещенными символами табуляции. Чтобы выполнить испытания, сохраните этот фрагмент кода в файле с расширением «.html» и откройте его в веб-браузере:

```
<html>
<head>
<title>Change CSV delimiters from commas to tabs</title>
</head>

<body>
<p>Input:</p>
<textarea id="input" rows="5" cols="75"></textarea>

<p><input type="button" value="Replace" onclick="commasToTabs()"></p>

<p>Output:</p>
<textarea id="output" rows="5" cols="75"></textarea>

<script>
function commasToTabs() {
    var input = document.getElementById("input"),
        output = document.getElementById("output"),
        regex = /(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?/g,
        result = "",
        match;

    while (match = regex.exec(input.value)) {
        // Проверить значение обратной ссылки 1
        if (match[1] == ",") {
        // Добавить в результат символ табуляции (на место
        // совпавшей запятой) и содержимое обратной ссылки 2.
        // Если обратная ссылка 2 не определена (когда из-за
        // необязательности вторая сохраняющая группа не участвовала
        // в сопоставлении), использовать пустую строку.
        result += "\t" + (match[2] || "");
    } else {
        // Добавить все совпадение в результат
        result += match[0];
    }
    // Предотвратить попадание некоторых браузеров в бесконечный цикл
```

```
            if (match.index == regex.lastIndex){
                regex.lastIndex++;
            }
        }
        output.value = result;
}
</script>
</body>
</html>
```

Обсуждение

Подход, описанный в этом рецепте, позволяет передавать каждое отдельное поле CSV (включая внутренние разрывы строк, экранированные кавычки и запятые) по одному за раз. Благодаря этому каждое последующее сопоставление будет начинаться с позиции непосредственно перед следующим разделителем полей.

Первая сохраняющая группа в регулярном выражении, `(,|\r?\n|^)`, совпадает с запятой, разрывом строки или с позицией в начале испытуемого текста. Так как механизм регулярных выражений испытывает варианты выбора слева направо, эти варианты перечислены в порядке уменьшения вероятности встречи с ними в файле CSV. Эта сохраняющая группа – единственная часть регулярного выражения, требующая обязательного совпадения. То есть возможна ситуация, когда общее совпадение с регулярным выражением будет пустой строкой, поскольку якорь `^` всегда совпадает один раз. Значение, совпавшее с этой первой сохраняющей группой, должно быть проверено в программном коде за пределами регулярного выражения, который замещает запятые требуемым разделителем (например, символом табуляции).

Мы еще не достигли конца регулярного выражения, но подход, описанный до сих пор, уже начинает казаться замысловатым. Может возникнуть вопрос – почему бы не написать регулярное выражение, совпадающее только с запятыми, которые необходимо заменить символами табуляции? Если бы было сделано именно так, простая замена всех совпадений помогла бы избавиться от программного кода за пределами регулярного выражения, проверяющего, совпадает ли сохраняющая группа 1 с запятой или какими-либо другим текстом. В конце концов, определить, находится ли запятая внутри или за пределами поля CSV в кавычках, можно с помощью опережающей и ретроспективной проверки, разве не так?

К сожалению, для реализации такого регулярного выражения потребовалась бы ретроспективная проверка бесконечной длины, чтобы точно определить, какие запятые находятся за пределами полей в кавычках, что возможно только в диалекте .NET (описание различных ограничений ретроспективной проверки можно найти в разделе «Различные уровни ретроспективной проверки» в рецепте 2.16). Однако даже разработчикам для платформы .NET следует избегать решений, основанных

на проверках соседних символов, потому что это может приводить к существенному усложнению и снижению производительности регулярного выражения.

Возвращаясь назад к описанию принципа действия регулярного выражения, заметим, что большая часть его заключена в следующую пару круглых скобок – сохраняющую группу 2. Эта вторая группа совпадает с единственным полем CSV, включая окружающие его кавычки. В отличие от предыдущей группы, эта группа является необязательной и может совпадать с пустыми полями.

Обратите внимание, что группа 2 содержит два альтернативных подвыражения, разделенных метасимволом ‹|›. Первая альтернатива, ‹[^",\r\n]+›, – это инвертированный символьный класс, за которым следует квантификатор «один или более раз» (‹+›), а все вместе это соответствует полю, не заключенному в кавычки. Чтобы соответствовать этому подвыражению, поле не должно содержать кавычки, запятые или разрывы строк.

Вторая альтернатива внутри группы 2, ‹"(?:[^"]|"")*"›, соответствует полю, заключенному в кавычки. Точнее, она соответствует символу кавычки, за которой следует ноль или более символов, не являющихся кавычкой, и повторяющихся (экранированных) кавычек, за которыми следует закрывающая кавычка. Квантификатор ‹*› в конце внутренней несохраняющей группы повторяет сопоставление двух внутренних вариантов выбора максимально возможное число раз, пока не будет найдена кавычка, не дублированная и потому завершающая поле.

Предположим, что выражение применяется к допустимому файлу CSV. Тогда первое совпадение, обнаруженное регулярным выражением, будет найдено в самом начале испытуемого текста, а каждое последующее должно начинаться непосредственно за концом предыдущего совпадения.

См. также

Рецепт 9.12, где описывается, как использовать регулярное выражение из этого рецепта для извлечения полей CSV, принадлежащих определенному столбцу.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

9.12. Извлечение полей CSV из определенного столбца

Задача

Необходимо извлечь все поля из третьего столбца файла CSV.

Решение

Для обхода всех полей в испытуемом тексте CSV можно повторно использовать регулярное выражение из рецепта 9.11. Добавив немного программного кода, можно подсчитывать поля в каждой строке, или *записи*, и извлекать поля только из определенной позиции.

Следующее регулярное выражение (показано в обычном режиме и в режиме свободного форматирования) совпадает с единственным полем CSV и предшествующим ему разделителем, сохраняя их в двух отдельных сохраняющих группах. Так как внутри полей в кавычках могут появляться разрывы строк, простой поиск от начала до конца в каждой строке CSV может не давать точных результатов. Сопоставляя и перешагивая через поля по одному, легко определить, какие разрывы строк находятся за пределами полей в кавычках, и благодаря этому начинать отсчет полей в новой записи.

Регулярные выражения в этом рецепте предназначены для работы только с допустимыми файлами CSV с точки зрения правил, которые приводились в описании формата «Значения, разделенные запятыми (Comma-Separated Values, CSV)» в начале этой главы.

```
(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
( , | \r?\n | ^ )  # Сохраняющая группа 1 совпадает с разделителями полей
                   # или с началом строки
(                  # Сохраняющая группа 2 совпадает с единственным полем:
  [^",\r\n]+       #   не заключенным в кавычки
|                  # Или:
  " (?:[^"]|"")* " #   заключенным в кавычки (может содержать
                   # экранированные кавычки)
)?                 # Группа является необязательной, так как
                   # поле может быть пустым
```

Параметры: режим свободного форматирования,
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Это в точности те же самые регулярные выражения, которые приводились в рецепте 9.11, и они могут использоваться для решения широкого круга других задач обработки файлов CSV. Следующий фрагмент про-

граммного кода демонстрирует, как можно использовать версию регулярного выражения не в режиме свободного форматирования для извлечения столбца CSV.

Пример веб-страницы с кодом на JavaScript

Следующий ниже фрагмент представляет законченную веб-страницу, включающую два многострочных поля ввода и кнопку с надписью Extract Column 3 (Извлечь столбец 3) между ними. После щелчка на кнопке из первого текстового поля Input (Вход) извлекается введенная в него строка, а затем с помощью регулярного выражения, продемонстрированного выше, из каждой записи извлекается значение третьего поля, после чего весь столбец (где значения полей отделяются разрывами строк) помещается в поле Output (Выход). Чтобы выполнить испытания, сохраните этот фрагмент кода в файле с расширением «.html» и откройте его в веб-браузере:

```
<html>
<head>
<title>Extract the third column from a CSV string</title>
</head>

<body>
<p>Input:</p>
<textarea id="input" rows="5" cols="75"></textarea>

<p><input type="button" value="Extract Column 3"
    onclick="displayCsvColumn(2)"></p>

<p>Output:</p>
<textarea id="output" rows="5" cols="75"></textarea>

<script>
function displayCsvColumn(index) {
    var input = document.getElementById("input"),
        output = document.getElementById("output"),
        columnFields = getCsvColumn(input.value, index);

    if (columnFields.length > 0) {
        // Вывести каждую запись в отдельной строке, отделив ее от других
        // записей символом перевода строки (\n)
        output.value = columnFields.join("\n");
    } else {
        output.value = "[No data found to extract]";
    }
}

// Вернуть массив полей из переданного текста CSV,
// индексация элементов в котором начинается с нуля
function getCsvColumn(csv, index) {
    var regex = /(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?/g,
```

```
        result = [],
        columnIndex = 0,
        match;

    while (match = regex.exec(csv)) {
        // Проверить значение обратной ссылки 1. Если это запятая,
        // увеличить значение column_index. Иначе сбросить его в ноль.
        if (match[1] == ",") {
            columnIndex++;
        } else {
            columnIndex = 0;
        }
        if (columnIndex == index) {
            // Добавить поле (обр. ссылка 2) в конец массива результата
            result.push(match[2]);
        }

        // Предотвратить попадание некоторых браузеров в бесконечный цикл
        if (match.index == regex.lastIndex){
            regex.lastIndex++;
        }
    }
    return result;
}
</script>
</body>
</html>
```

Обсуждение

Так как здесь используется регулярное выражение из рецепта 9.11, мы не будем снова подробно описывать, как оно действует. Однако этот рецепт включает новый пример программного кода на JavaScript, где данное регулярное выражение используется для извлечения полей с определенным порядковым номером из каждой записи CSV в испытуемом тексте.

Функция `getCsvColumn()` в представленном коде выполняет итерации через совпадения в испытуемом тексте. После каждого совпадения проверяется содержимое обратной ссылки 1. Если она содержит запятую, следовательно, было найдено совпадение не с первым полем в записи, поэтому производится наращивание переменной `columnIndex`, где хранится порядковый номер текущего столбца. Если обратная ссылка 1 содержит нечто отличное от запятой (то есть пустую строку или разрыв строки), следовательно, было найдено совпадение с первым полем в новой строке, поэтому значение переменной `columnIndex` сбрасывается в ноль.

Затем программный код проверяет, не достиг ли счетчик `columnIndex` индекса искомого столбца, который требуется извлечь. Каждый раз когда проверка дает положительный результат, в массив с результатами добавляется значение обратной ссылки 2 (все, что следует за разделителем полей). После того как будет выполнен обход всего испытуемого

текста, функция `getCsvColumn()` вернет массив, содержащий значения всего заданного столбца (третьего в этом примере). Список совпадений записывается во второе текстовое поле на странице, при этом каждое значение отделяется символом перевода строки (`\n`).

Этот пример можно было бы немного улучшить, дав пользователю возможность указывать номер столбца, который следует извлечь, с помощью дополнительного поля ввода или посредством запроса. Обсуждаемая функция `getCsvColumn()` написана в предположении такой возможности и позволяет указывать индекс желаемого столбца, отсчет которых начинается с нуля, в виде второго аргумента (`index`).

Варианты

Хотя использование программного кода для обхода текста по одному полю CSV за раз обеспечивает дополнительную гибкость, при использовании текстового редактора может оказаться вполне достаточным ограничиться операцией поиска с заменой. В этой ситуации можно добиться похожих результатов, производя замещение совпадения с каждой полной записью значением поля с требуемым индексом (с помощью обратной ссылки). Следующие регулярные выражения иллюстрируют этот прием для различных индексов столбцов, замещая каждую запись полем из заданного столбца.

Все эти регулярные выражения не будут совпадать с записями, число полей в которых меньше номера искомого столбца, поэтому такие записи останутся в неизменном виде.

Соответствует записи CSV и сохраняет поле из столбца с номером 1 в обратной ссылке 1

```
^([^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует записи CSV и сохраняет поле из столбца с номером 2 в обратной ссылке 1

```
^(?:[^",\r\n]+|"(?:[^"]|"")*")?,([^",\r\n]+|"(?:[^"]|"")*")?↵
(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Соответствует записи CSV и сохраняет поле из столбца с номером 3 или выше в обратной ссылке 1

```
^(?:[^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?){1},↵
([^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Увеличив число в квантификаторе ‹{1}›, можно заставить регулярное выражение обрабатывать поля с порядковыми номерами выше 3. Например, если изменить значение квантификатора на ‹{2}›, будут сохраняться поля из столбца с номером 4, если изменить на ‹{3}› – будут сохраняться поля из столбца с номером 5 и т. д. Если предполагается работать с полями из столбца 3, квантификатор ‹{1}› можно просто удалить, так как в этом случае он не оказывает никакого воздействия.

Замещающий текст

Со всеми этими регулярными выражениями используется один и тот же замещающий текст (обратная ссылка 1). В результате замещения каждого совпадения содержимым обратной ссылки 1 должны остаться только искомые поля.

```
$1
```

Диалекты замещающего текста: .NET, Java, JavaScript, Perl, PHP

```
\1
```

Диалекты замещающего текста: Python, Ruby

См. также

Рецепт 9.11, где демонстрируется, как с применением регулярного выражения в этом рецепте заменить запятые, играющие роль разделителей полей в файле CSV, символами табуляции.

Приемы, использовавшиеся в регулярных выражениях и в замещающем тексте в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.8 описывается применение оператора выбора. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов. В рецепте 2.21 демонстрируется, как вставлять текст, совпавший с сохраняющей группой, в текст замены.

9.13. Сопоставление с заголовком раздела в файле INI

Задача

Необходимо отыскать совпадения со всеми заголовками разделов в файле INI.

Решение

Заголовки разделов в файлах INI находятся в начале строки текста и окружены квадратными скобками (например, `[Section1]`). Эти правила легко оформить в виде регулярного выражения:

```
^\[[^\]\r\n]+]
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

В этом регулярном выражении немного частей, поэтому его легко будет разобрать:

- Благодаря включенному режиму «символам ^ и $ соответствуют границы строк» начальный символ ‹^› совпадает с началом каждой строки в тексте.
- ‹\[› совпадает с литералом [. Чтобы символ [не интерпретировался как начало символьного класса, он экранируется обратным слэшем.
- ‹[^\]\r\n]› – это инвертированный символьный класс, которому соответствует любой символ, кроме], возврата каретки (\r) и перевода строки (\n). Вслед за классом сразу же следует квантификатор ‹+›, позволяющий классу совпадать с одним или более символами, пока не будет обнаружено совпадение с...
- Завершающему символу ‹]› соответствует литерал] в конце заголовка раздела. Этот символ не требуется экранировать, так как он не находится внутри символьного класса.

Варианты

Если требуется просто отыскать определенный заголовок раздела, решение выглядит еще проще. Следующее регулярное выражение совпадает с заголовком раздела `Section1`:

```
^\[Section1]
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Единственное отличие от сопоставления с простым текстом «[Section1]» состоит в том, что совпадение должно находиться в начале строки текста. Это предотвращает возможность совпадения с закомментированным заголовком раздела (которому предшествует символ точки с запятой) или с фрагментом, который выглядит как заголовок, но фактически является частью значения параметра (например, `Item1=[Value1]`).

См. также

Рецепт 9.14, где описывается, как обеспечивать совпадение с разделами в файлах INI. Рецепт 9.15, где описывается, как добиться того же самого в отношении пар имя-значение.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

9.14. Сопоставление с разделом в файле INI

Задача

Требуется обеспечить совпадение со всем разделом в файле INI (то есть с заголовком раздела и со всеми парами имя-значение в этом разделе), чтобы разбить файл INI на блоки или для обработки блоков по отдельности.

Решение

В рецепте 9.13 показано, как обеспечить совпадение с заголовком раздела в файле INI. Чтобы обеспечить совпадение со всем разделом, мы начнем сопоставление точно так же, как в предыдущем рецепте, но будем продолжать сопоставление, пока не будет достигнут конец текста или символ [в начале строки (так как это указывает на начало нового раздела):

```
^\[[^\]\r\n]+](?:\r?\n(?:[^[\r\n].*)?)*
```

Параметры: символам ^ и $ соответствуют границы строк (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Или в режиме свободного форматирования:

```
^ \[ [^\]\r\n]+ ] # Совпадение с заголовком раздела
(?:               # Далее следует остальная часть раздела:
  \r?\n           #   Последовательность символов разрыва строки
  (?:             #   После каждого начала строки совпадает:
    [^[\r\n]      #     С любым символом, кроме "[" и разрыва строки,
    .*            #     И с остатком строки
  )?              # Необязательная, чтобы совпадать с пустыми строками
)*                # Продолжать, пока не будет найден конец раздела
```

Параметры: символам ^ и $ соответствуют границы строк, режим свободного форматирования (режим «точке соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Обсуждение

Это регулярное выражение начинается сопоставлением заголовка раздела в файле INI с шаблоном `‹^\[[^\]\r\n]+]›` и продолжает построчное сопоставление, пока не будет встречена строка, начинающаяся с символа [. Рассмотрим следующий текст:

```
[Section1]
Item1=Value1
Item2=[Value2]

; [SectionA]
; Заголовок раздела SectionA был закомментирован

ItemA=ValueA ; ItemA не закомментирован и является частью раздела Section1

[Section2]
Item3=Value3
Item4 = Value4
```

В данном тексте это регулярное выражение обнаружит два совпадения. Первое будет простираться от начала текста до заголовка `[Section2]`, включая пустую строку перед ним. Второе совпадение будет простираться от начала заголовка Section2 до конца текста.

См. также

Рецепт 9.13, где описывается, как обеспечить совпадение с заголовком раздела в файле INI. Рецепт 9.15, где описывается, как добиться совпадения с парами имя-значение.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.1 описывается, какие специальные символы следует экранировать. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

9.15. Сопоставление с парами имя-значение в файле INI

Задача

Требуется обеспечить совпадение с парами имя-значение в файле INI (например, `Item1=Value1`), разбивая каждое совпадение на две части с помощью сохраняющих групп. Обратная ссылка 1 должна хранить имя параметра (`Item1`), а обратная ссылка 2 – значение (`Value1`).

Решение

Ниже приводится регулярное выражение, выполняющее эту работу:

```
^([^=;\r\n]+)=([^;\r\n]*)
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

То же самое регулярное выражение в режиме свободного форматирования:

```
^                 # Начало строки текста
( [^=;\r\n]+ )    # Сохранить имя в обратной ссылке 1
=                 # Разделитель между именем и значением
( [^;\r\n]* )     # Сохранить значение в обратной ссылке 2
```

Параметры: символам ^ и $ соответствуют границы строк, режим свободного форматирования
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Обсуждение

Как и в других рецептах этой главы, имеющих отношение к файлам INI, здесь мы работает с весьма простыми составляющими. Регулярное выражение начинается с символа ‹^›, который обеспечивает совпадение с началом строки в тексте (при этом должен быть включен режим «символам ^ и $ соответствуют границы строк»). Очень важно, чтобы совпадение начиналось с начала строки, потому что в противном случае можно было бы найти совпадение с частью закомментированной строки.

Затем регулярное выражение использует сохраняющую группу, содержащую инвертированный символьный класс ‹[^=;\r\n]›, за которым следует квантификатор «один или более раз» ‹+›, что обеспечивает совпадение с именем параметра и сохранение его в виде обратной ссылки 1. Инвертированному классу соответствует любой символ, кроме следующих четырех: знак равенства, точка с запятой, возврат каретки (‹\r›) и перевод строки (‹\n›). Символы возврата каретки и перевода строки используются как признак конца параметра в файле INI, точка с запятой отмечает начало комментария, а знак равенства отделяет имя параметра от значения.

После сопоставления с именем параметра далее в регулярном выражении следуют сопоставление с литералом знака равенства (разделитель имени и значения) и затем со значением параметра. Совпадение со значением обеспечивает вторая сохраняющая группа, в которой используется шаблон, похожий на шаблон имени параметра, но имеющий на два ограничения меньше. Во-первых, этот второй шаблон допускает совпадение со знаком равенства как частью значения (то есть инвертированный символьный класс на один символ короче). Во-вторых, здесь используется квантификатор ‹*›, чтобы устранить необходимость соот-

ветствовать хотя бы одному символу (так как, в отличие от имен, значения могут быть пустыми).

Вот и все.

См. также

Рецепт 9.13, где описывается, как обеспечить совпадение с заголовком раздела в файле INI. Рецепт 9.14, где описывается, как обеспечивать совпадение с разделами в файлах INI.

Приемы, использовавшиеся в регулярных выражениях в этом рецепте, обсуждаются в главе 2. В рецепте 2.2 объясняется, как искать совпадения с непечатаемыми символами. В рецепте 2.3 рассказывается о символьных классах. В рецепте 2.5 обсуждаются якорные метасимволы. В рецепте 2.9 рассказывается о группировке. В рецепте 2.12 объясняется, как организовать сопоставление с повторяющимися комбинациями символов.

Алфавитный указатель

Символы

7-битный набор символов, 55
& (амперсанд), экранирование, 452
| (вертикальная черта)
  как оператор выбора, 90, 329, 405
  экранирование, 451
!~, оператор (Perl), 185
- (дефис)
  диапазоны внутри символьных классов, 57
  как метасимвол, 50
  удаление из номера кредитной карты, 387, 390
  экранирование, 451
, (запятая)
  как разделитель групп разрядов, 483
  экранирование, 452
* (звездочка)
  как квантификатор, 305, 502, 503
  как максимальный квантификатор, 108
  как метасимвол, 50
  экранирование, 451
? (знак вопроса)
  как минимальный квантификатор, 108, 503
  как квантификатор ноль или один раз, 105
  как метасимвол, 50
  экранирование, 451
(?:), несохраняющая группа, 305
(?m), модификатор режима, 70
$ (знак доллара)
  как метасимвол, 50
  как якорный метасимвол, 66, 67, 69, 309, 157
  экранирование, 451
    в замещающем тексте, 134
$`, переменная, 143, 203, 266
$_, переменная, 143
$', переменная, 143, 266
$&, переменная, 137, 198, 203
$~, переменная (Ruby), 198
  сохраняющие группы, 210
$~[0], переменная, 198
+ (знак плюс)
  для обозначения захватывающего квантификатора, 111
  как квантификатор, 305
    и возвраты, 507
  как метасимвол, 50
  экранирование, 451
[] (квадратные скобки)
  для определения символьных классов, 57
  как метасимволы, 50
  символьные классы, 305, 377, 503
  экранирование, 450
@ (коммерческое at)
  представление в Perl, 157
  совпадение, 304
@+, переменная, 203
@–, переменная, 203
() (круглые скобки)
  атомарная группировка, 114, 348
  как метасимволы, 50
  повторный поиск соответствия с ранее совпавшим текстом, 96
  сохраняющая группировка, 92
  экранирование, 450
^ (крышка)
  как метасимвол, 50
  как якорный метасимвол, 66, 67, 68, 305, 309
  отрицание внутри символьных классов, 57
  экранирование, 451
\ (обратный слэш)
  в исходных текстах, 152
  внутри символьных классов, 57
  как метасимвол, 50
  как символ экранирования, 50
  экранирование, 448, 451
    в замещающем тексте, 134

=~, оператор
Perl, 185
Ruby, 186, 204
(решетка)
оформление комментариев, 131
экранирование, 452
. (точка), метасимвол, 510
злоупотребление, 64
как метасимвол, 50
соответствует любому символу, 62
экранирование, 451
{} (фигурные скобки)
для организации повторений, 104
как метасимволы, 50
экранирование, 450
%+, хеш, 213, 250

A

\A, якорный метасимвол, 66, 67, 68, 306
ActionScript, язык программирования, 148
appendReplacement(), метод, 255
appendTail(), метод, 255

B

\b, метасимвол, граница слова, 60, 71, 312, 329, 401, 497, 498
\B, не граница слова, 72
<b>, тег, замещение тегом , 631
begin(), метод, Ruby (класс MatchData), 204

C

C, язык программирования, 148
\cA до \cZ, значения, 54
C++, язык программирования, 148
C#, язык программирования, 145
импортирование библиотеки регулярных выражений, 161
литералы регулярных выражений, 154
создание объектов регулярных выражений, 165
CacheSize, свойство (.NET), 165
CANON_EQ, константа (Java), 176
CASE_INSENSITIVE, константа (Java), 173
cellspacing, атрибут, добавление в теги <table>, 658
CIL (Common Intermediate Language – общий промежуточный язык), 170
Comma-Separated Values (значения, разделенные запятыми, CSV), 609
извлечение полей из определенного столбца, 677
изменение разделителя, 672
COMMENTS, константа (Java), 172
compile(), метод
Java (класс Pattern), 155, 166, 172
Python (модуль re), 168
параметры регулярных выражений, 174
Ruby (класс Regexp), 169
Count, свойство (Match.Groups), 208
CSV (Comma-Separated Values – значения, разделенные запятыми), 609
Currency, блок Юникода, 83

D

\d, метасимвол, 58, 456
\D, метасимвол, 58
Delphi, язык программирования, 149
Delphi Prism, язык программирования, 150
DOCTYPE, объявление (HTML), 606
DOTALL, константа (Java), 172

E

\E, лексема, для подавления метасимволов, 51
ECMA-262, стандарт, 146, 148
ECMAScript, значение (RegexOptions), 176
ECMAScript, язык программирования, 146
end(), метод
Java (класс Matcher), 202, 209
Python (класс MatchObject), 203
Ruby (класс MatchData), 204
EPP (Extensible Provisioning Protocol – расширяемый протокол представления информации), 316
ereg, семейство функций (PHP), 147
литералы регулярных выражений, 156
exec(), метод
XRegExp, 225
JavaScript, 202, 209
обход всех совпадений в цикле, 224
ExplicitCapture, значение (RegexOptions), 176
EXTENDED, константа (Regexp), 175
Extensible Hypertext Markup Language (расширяемый язык разметки гипертекста, XHTML), 607
добавление атрибутов в теги <table>, 658
поиск тегов, 612
удаление всех тегов, за исключением выбранных, 635
Extensible Markup Language (расширяемый язык разметки, XML), 608
декодирование мнемоник, 650
добавление атрибутов в теги <table>, 658

поиск определенных атрибутов, 653
поиск слов в комментариях, 667
поиск тегов, 612, 615
сопоставление с именами, 639
удаление всех тегов, за исключением выбранных, 635
удаление комментариев, 662

F

find(), метод, Java (класс Matcher), 184, 197, 202, 216
обход всех совпадений в цикле, 223
findall(), метод, Python (модуль re), 218
finditer(), метод, Python (модуль re), 226
forEach, метод (XRegExp), 225

G

Get-Unique, команда (PowerShell), 433
grep, функция, поддержка в проекте R, 151
Groovy, язык программирования, 150
Group, класс (.NET), 208
group(), метод
Python (модуль re), 210
Java (класс Matcher), 209
GroupCollection, класс (.NET), 208
groupdict(), метод, Python (класс MatchObject), 210
Groups, свойство .NET (функция Match()), 208
groups(), метод, Python (класс MatchObject), 210
gsub(), метод, Ruby (класс String), 244, 257
обратные ссылки, 248

H

Hypertext Markup Language (язык разметки гипертекста, HTML), 604
добавление атрибутов в теги <table>, 658
добавление тегов <p> в простой текст, 646
замещение тега <b> тегом , 631
поиск тегов, 612
удаление всех тегов, за исключением выбранных, 635

I

IGNORECASE, константа (Regexp), 175
IgnoreCase, параметр (RegexOptions), 172
IgnorePatternWhitespace, параметр (RegexOptions), 172
Index, свойство (.NET), 201
index, свойство (JavaScript), 202
IndexOutOfBoundsException, исключение, 242
INI, файлы
сопоставление с заголовком раздела, 681
сопоставление с парами имя-значение, 684
сопоставление с разделом, 683
IPv4, адреса, сопоставление, 565
IPv6, адреса, сопоставление, 569
IsMatch(), метод (.NET), 184
ISO 8601, форматы, 330

J

Java, язык программирования, 145
импортирование библиотеки регулярных выражений, 161
литералы регулярных выражений, 155
определение замещающего текста, 24
параметры регулярных выражений, 172, 176
поддержка регулярных выражений, 20
создание объектов регулярных выражений, 165
JavaScript, язык программирования, 146
библиотека XRegExp, 146
библиотека регулярных выражений, 161
литералы регулярных выражений, 156
определение замещающего текста, 24
параметры регулярных выражений, 173, 176
поддержка регулярных выражений, 20, 146
создание объектов регулярных выражений, 167
java.util.regex, пакет, для использования регулярных выражений, 161

K

\K, метасимвол, 513
ksort(), функция (PHP), 243

L

lastIndex, свойство, JavaScript (класс RegExp), 202, 224, 418
Length, свойство (.NET), 201
length, свойство (JavaScript), 202
length(), метод, Ruby (класс MatchData), 204

M

m//, оператор (Perl), 185, 198
сохраняющие группы, 210

Match, класс (.NET)
NextMatch(), метод, 222
свойства Index и Length, 201
match(), метод
Ruby (класс Regexp), 204
JavaScript, 216
Match(), метод (.NET), 201
Groups, свойство, 208
Value, свойство, 196
обход всех совпадений в цикле, 222
MatchData, класс (Ruby), 198
begin(), метод, 204
end(), метод, 204
length(), метод, 204
offset(), метод, 204
size(), метод, 204
Matcher, класс (Java), 167
end(), метод, 202, 209
find(), метод, 184, 197, 202, 216
обход всех совпадений в цикле, 223
group(), метод, 209
reset(), метод, 167, 197
start(), метод, 202, 209
Matches(), метод (.NET), 215
MatchEvaluator, класс (.NET), 253
MatchObject, класс (Python)
end(), метод, 203
start(), метод, 203
mb_ereg, семейство функций (PHP), 147
литералы регулярных выражений, 156
Microsoft VBScript, библиотека, 152
MULTILINE, константа (Java), 172
MULTILINE, константа (Regexp), 175
Multiline, параметр (RegexOptions), 172

N

\n (символ перевода строки),
представление в
C#, 155
Java, 155
Python, 153, 159
NEAR-поиск, 421
.NET, платформа
параметры регулярных выражений, 172, 175
создание объектов регулярных выражений, 164
new(), метод (Ruby), 169, 175
Nregex, веб-приложение для тестирования регулярных выражений, 35

O

offset(), метод, Ruby (класс MatchData), 204
Onigurama, библиотека, 148

P

\p{}, определение категории Юникода, 76
\P{}, для проверки категории Юникода, 80
Pattern, класс (Java), 165
compile(), метод, 155, 166, 172
split(), метод, 271
константы параметров, 172, 176
PatternSyntaxException, исключение, 166
PCRE (Perl-Compatible Regular Expressions, Perl-совместимые регулярные выражения), библиотека, 21, 147
определение замещающего текста, 24
Perl, язык программирования, 147
библиотека регулярных выражений, 162
литералы регулярных выражений, 157
определение замещающего текста, 24
параметры регулярных выражений, 174, 177
поддержка регулярных выражений, 18, 147
создание объектов регулярных выражений, 168
PHP, язык программирования, 146
импортирование библиотеки регулярных выражений, 162
литералы регулярных выражений, 156
определение замещающего текста, 24
параметры регулярных выражений, 174, 177
создание объектов регулярных выражений, 167
функции регулярных выражений, 146
POSIX-совместимые механизмы регулярных выражений, 472
PowerGREP, приложение, 44
PowerShell, язык сценариев, 150
preg, семейство функций (PHP), 147
preg_match_all(), функция, 217, 226
preg_match(), функция, 185, 198
обход всех совпадений в цикле, 225
preg_replace_callback(), функция, 256
preg_replace(), функция, 147, 242
именованные сохраняющие группы, 250
обратные ссылки, 247
preg_split(), функция, 273
и сохраняющая группировка, 278
доступность, 162
литералы регулярных выражений, 156
параметры регулярных выражений, 174
создание объектов регулярных выражений, 168

PREG_OFFSET_CAPTURE, константа, 203, 209, 217
PREG_PATTERN_ORDER, константа, 217
PREG_SET_ORDER, константа, 217
PREG_SPLIT_DELIM_CAPTURE, константа, 278
PREG_SPLIT_NO_EMPTY, константа, 273, 278
Python, язык программирования, 147
  импортирование библиотеки регулярных выражений, 162
  литералы регулярных выражений, 158
  определение замещающего текста, 25
  параметры регулярных выражений, 174, 178
  поддержка регулярных выражений, 22
  создание объектов регулярных выражений, 168
  функции регулярных выражений, 147

Q

\Q, лексема, для подавления метасимволов, 51
qr//, оператор (Perl), 168

R

R, проект, 151
re, модуль (Python), 25, 147, 158
  compile(), функция, 168
  findall(), метод, 218
  finditer(), метод, 226
  group(), метод, 210
  search(), функция, 185, 198
  split(), метод, 274
    и сохраняющая группировка, 279
  sub(), метод, 244, 257
    обратные ссылки, 248
  импортирование, 162
  параметры регулярных выражений, 174
REALbasic, язык программирования, 151
Regex, класс (.NET)
  IsMatch(), метод, 184
  Matches(), метод, 215
  Replace(), метод, 240, 253, 254
    обратные ссылки, 247
  Split(), метод, 269
    обратные ссылки, 277
RegEx, класс (REALbasic), 151
Regex(), конструктор (.NET), 154, 165
  RegexOptions, перечисление (параметры регулярных выражений), 170
regex.larsolavtorvik.com, веб-приложение для тестирования регулярных выражений, 32
RegexOptions, перечисление, 172, 176
  Compiled, значение, 170
RegExp, класс (JavaScript)
  exec(), метод, 209
    обход всех совпадений в цикле, 224
  index, свойство, 202
  lastIndex, свойство, , 202, 224
  length, свойство, 202
  test(), метод, 184
Regexp, класс (Ruby)
  compile(), метод, 169
  new(), метод, 169, 175
RegExp(), конструктор (JavaScript), 167
RegexPlanet, веб-приложение для тестирования регулярных выражений, 32
regexpr, функция, поддержка в проекте R, 151
RegexRenamer, приложение, 46
replace, функция, класс String
  удаление пробельных символов, 445
replace(), метод, JavaScript (класс String), 242
  обратные ссылки, 247
Replace(), метод (.NET), 240, 253, 254
  обратные ссылки, 247
replaceAll(), метод, Java (класс String), 241
  обратные ссылки, 247
replaceFirst(), метод, Java (класс String), 241
  обратные ссылки, 247
reset(), метод, Java (класс Matcher), 167, 197
RFC 3986, стандарт, 543
RFC 4180, стандарт, 610
Ruby, язык программирования, 147
  библиотека регулярных выражений, 162
  литералы регулярных выражений, 159
  определение замещающего текста, 25
  параметры регулярных выражений, 175, 178
  поддержка регулярных выражений, 22
  создание объектов регулярных выражений, 169
  функции регулярных выражений, 147

S

s///, оператор (Perl), 243
  /e, модификатор, 257
  обратные ссылки, 247
\s, метасимвол, 344, 345
  как пробельный символ, 60

\S, метасимвол, 304, 344
Scala, язык программирования, 151
scala.util.matching, пакет, 151
scan(), метод, Ruby (класс String), 226
<script>, элементы (HTML), 605
search(), функция, Python (модуль re), 185, 198
Singleline, параметр (RegexOptions), 172
size(), метод, Ruby (класс MatchData), 204
split(), метод, Java (класс String), 271
split(), метод, JavaScript (класс String), 272
Split(), метод, .NET, 269
  обратные ссылки, 277
split(), метод, Python (модуль re), 274
  и сохраняющая группировка, 279
split(), метод, Ruby (класс String), 274
  и сохраняющая группировка, 280
split(), метод, XRegExp (класс XRegExp), 272
split(), функция (Perl), 273
  и сохраняющая группировка, 279
start(), метод, Java (класс Matcher), 202, 209
start(), метод, Python (класс MatchObject), 203
String, класс (Java)
  replaceAll(), метод, 241
    обратные ссылки, 247
  replaceFirst(), метод, 241
    обратные ссылки, 247
String, класс (JavaScript)
  match(), метод, 197, 216
  replace(), метод, 242
    обратные ссылки, 247
  replace, функция, класс String
    удаление пробельных символов, 445
  split(), метод, 272
String, класс (Ruby)
  gsub(), метод, 244, 257
    обратные ссылки, 248
  scan(), метод, 226
  split(), метод, 274
    и сохраняющая группировка, 280
strlen(), функция, 203
<style>, элементы (HTML), 605
sub, функция, поддержка в проекте R, 151
sub(), метод, Python (модуль re), 25, 244, 257
  обратные ссылки, 248
System.Text.RegularExpressions.Regex, класс, 165

T

<table>, тег, добавление атрибутов, 658
test(), метод (JavaScript), 184
TPerlRegEx, компонент, 149

U

\u, метасимвол для представления кодовых пунктов Юникода
  в строках Java, 155
UNC, пути, 586, 591, 595
UNICODE (или U), флаг, 59, 341
UNICODE_CASE, константа (Java), 173
uniq, утилита (UNIX), 433
UNIX_LINES, константа (Java), 176
URL, адреса
  извлечение имени пользователя, 548
  извлечение имени хоста, 551
  извлечение номера порта, 553
  извлечение пути, 556
  извлечение строки запроса, 560
  извлечение схемы, 546
  извлечение фрагмента, 561
  поиск в тексте, 529
    заключенных в кавычки, 532
    заключенных в скобки, 533
  проверка, 525
  универсальные, проверка, 540

V

Value, свойство, .NET (класс Match), 196
VB.NET, язык программирования, 145, 151
  импортирование библиотеки регулярных выражений, 161
  литералы регулярных выражений, 155
  создание объектов регулярных выражений, 165
Visual Basic 6, язык программирования, 151

W

\w, метасимвол, символ слова, 59, 73, 304, 497
  ограничение числа слов, 345
\W, метасимвол, 59, 73
  ограничение числа слов, 345
Windows Grep, приложение, 45

X

XML 1.0, спецификация, 640, 641, 643
XML 1.1, спецификация, 640, 641, 645
XRegExp, библиотека
  импортирование, 161
  литералы регулярных выражений, 156
  определение замещающего текста, 24
  параметры регулярных выражений, 173, 176

поддержка регулярных выражений, 20, 146
создание объектов регулярных выражений, 167
XRegExp, класс (XRegExp)
exec, метод, 225
forEach, метод, 225
split(), метод, 272

Z

\z, якорный метасимвол, 66, 67
\Z, якорный метасимвол, 66, 67, 69, 306

A

алгоритм Луна, 392
алфавитно-цифровые символы, ограничение ввода, 338
алфавиты Юникода, 76, 84
перечисление всех символов, 88
атомарная группировка, 114, 116, 348
в сопоставлении со словами из списка, 495
атомарность опережающей и ретроспективной проверок, 123, 353, 429
атрибуты (языки разметки), 605
белые списки, 638
поиск определенных атрибутов, 653

Б

белые списки, 638
бесконечная длина совпадения с ретроспективной проверкой, 417
бесконечное число повторений, 105
библиотеки регулярных выражений, импортирование, 160
блоки Юникода, 76, 80
перечисление всех символов, 88
буквы устройств, извлечение из путей в Windows, 594

В

ведущие нули, удаление, 465
ведущие пробельные символы, удаление, 442
вещественные числа, 479
добавление разделителей групп разрядов, 484
вложенные квантификаторы, 665
вложенные символьные классы, 61
возвраты, 108
бесполезные, устранение, 111
и квантификатор +, 507
катастрофические, 116
управление, 623
восьмеричные числа, поиск, 463
время в традиционных форматах, проверка, 327
вставка контекста совпадения в замещающий текст, 142
вставка совпадения с регулярным выражением в замещающий текст, 136
вставка части совпадения с регулярным выражением в замещающий текст, 138
встроенные документы (here documents), 514
извлечение из исходного кода, 514
выбор, поиск слов из списка, 403
выделение лексем, 234

Г

гиперссылки, включение адресов URL, 536
границы слова, 71, 329, 497
при поиске чисел, 457
сопоставление в символьных классах, 60
графемы Юникода, 76, 85
группировка, 92
атомарная группировка, 114, 116, 348
и сохранение, 92
именованные сохранения, 140
имитация ретроспективной проверки, 126
повторение групп, 106
проверка соседних символов, 118, 344, 402
проверка условия, 127
пустые обратные ссылки, 425
ссылки на несуществующие группы, 140

Д

двоичные числа, поиск, 462
декодирование мнемоник XML, 650
десятичные числа
поиск, 464
преобразование римских чисел, 492
диакритические знаки, 86
диалекты определения замещающего текста, 23
диалекты регулярных выражений, 19, 148
диапазоны внутри символьных классов, 57
ДКА, механизм регулярных выражений, 472
длина совпадения, определение, 199
длина строки, проверка, 342
доменные имена, проверка, 562
дубликаты строк, удаление, 433

Е

европейские регистрационные номера плательщиков НДС, проверка, 393
единственная строка (single line), режим, 63

З

завершающие пробельные символы, удаление, 442
замена всех совпадений, 237
замена совпадений с повторным использованием частей совпадений, 245
замещающий текст
  вставка контекста совпадения, 142
  вставка совпадения с регулярным выражением, 136
  вставка части совпадения с регулярным выражением, 138
  сгенерированный в программном коде, 251
  экранирование символов, 134
замещение всех совпадений внутри совпадений с другим регулярным выражением, 258
замещение всех совпадений между совпадениями с другим регулярным выражением, 260
записи, в файлах CSV, 610
захватывающие квантификаторы, 111
знаки пунктуации
  категория Юникода, 79
значения, разделенные запятыми (Comma-Separated Values, CSV), 609
  извлечение полей из определенного столбца, 677
  изменение разделителя, 672

И

идентификатор пространства имен (Namespace Identifier, NID), 539
идентификаторы, извлечение из исходного кода, 498
извлечение из исходного кода
  всех комментариев, 504
  встроенных документов, 514
  идентификаторов, 498
  ключевых слов, 495
  литералов регулярных выражений, 511
  многострочных комментариев, 502
  однострочных комментариев, 502
  операторов, 500
  строк, 505
    с экранированными символами, 509
  числовых констант, 499
извлечение списка всех совпадений, 213
извлечение текста совпадения, 192
извлечение части совпавшего текста, 205
изменение разделителя, используемого в файлах CSV, 672
имена, в языке разметки XML, 609
  сопоставление, 639
имена папок, извлечение из путей в Windows, 596
имена файлов
  извлечение из путей в Windows, 599
  удаление недопустимых символов, 601
имена, форматирование, 373
именованное сохранение, 99, 211, 248
  группы с одинаковыми именами (.NET), 326
именованные обратные ссылки, 101
именованные сохранения, 140
имитация ретроспективной проверки, 416, 418
импортирование библиотеки регулярных выражений, 160
имя пользователя, извлечение из адреса URL, 548
имя сервера, извлечение из путей в Windows, 595
имя хоста, извлечение из адреса URL, 551
инструменты для работы с регулярными выражениями, 25
  Expresso, 39
  grep, 44
  myregexp.com, 37
  Nregex, 35
  RegexBuddy, 26
  regex.larsolavtorvik.com, 32
  RegexMagic, 30
  RegexPal, 29
  RegexPlanet, 32
  Rubular, 36
  SDL Regex Fuzzer, 42
  The Regulator, 40
  популярные текстовые редакторы, 47
интервальный квантификатор, 345
история термина «регулярное выражение», 17
исходные тексты, литералы регулярных выражений, 152

К

катастрофические возвраты, 116, 507
категории Юникода, 76, 78
квантификаторы, 310
  бесконечного числа повторений, 105
  вложенные, 665

переменного числа повторений, 104
фиксированного числа повторений, 104
ключевые слова, извлечение из исходного кода, 495
кодовые пункты, которым не присвоены символы (Юникод), 79
кодовые пункты Юникода, 75, 77
блоки, 80
комбинированные знаки, 85
комбинированный формат файлов журналов (Combined Log Format), 520
комментарии
в HTML, 606
в XML
поиск слов, 667
проверка, 663
удаление, 662
все, извлечение из исходного кода, 504
в регулярных выражениях, 131
в файлах INI, 611
многострочные, извлечение из исходного кода, 502
однострочные, извлечение из исходного кода, 502
компактная форма записи (IPv6), 572, 580
смешанная, 574, 576, 583, 584
конечная длина совпадения с ретроспективной проверкой, 417
конструкция выбора, поиск чисел в определенном диапазоне, 470
контекст совпадения, 142
контрольная сумма ISBN-10, 364
контрольная сумма ISBN-13, 365

Л

левый контекст, 143
литералы регулярных выражений
в исходных текстах, 152
извлечение из исходного кода, 511
литеральный текст, сопоставление, 50
Луна, алгоритм, 392

М

максимальное число повторений, 107
максимальные квантификаторы, 107
устранение бесполезных возвратов, 111
международные телефонные номера, проверка, 314
метасимволы, 51
внутри символьных классов, 57
экранирование, 448
механизмы, управляемые регулярными выражениями, 90
механизмы, управляемые текстом, 90
минимальное число повторений, 107
минимальные квантификаторы, 107, 108
устранение бесполезных возвратов, 111
мнемоники, 606, 647
декодирование, 650
мнемонические ссылки на символы, 606, 647
многострочный (multiline) режим, 69
многострочный текст
проверка числа строк, 348
модификаторы режима, используемые с несохраняющими группами, 95

Н

негативная опережающая проверка, 411, 413
негативная ретроспективная проверка, 415
недействительные ссылки, сообщения в файле журнала веб-сервера, 521
неотображаемые символы,
представление в Python, 159
несохраняющая группировка, 499
несохраняющие группы, 94
нечувствительность к регистру символов, режим, установка, 172
НКА, механизм регулярных выражений, 472
номера кредитных карт, проверка, 387
номера социального страхования, проверка, 356
нули, удаление ведущих, 465

О

обобщенный формат файлов журналов (Common Log Format), 516
обратные ссылки, 96
именованные, 99, 101
использование пустых обратных ссылок, 425
на несуществующие группы, 140
повторное использование частей совпадений, 245
поиск повторяющихся слов, 429
обход всех совпадений в цикле, 219
объединение символьных классов, 61
объекты регулярных выражений, создание, 162
объявление типа документа (HTML), 606
ограничение числа непробельных символов, 344
ограничение числа слов, 345
ограничение числа строк в тексте, 348
оператор выбора, 90
оператор подстановки (Perl), 157

оператор сопоставления с шаблоном (Perl), 157
операторы, извлечение из исходного кода, 500
опережающая проверка, 119, 344
  негативная, 411, 413
  отрицательная, 120
  положительная, 119
определение позиции вставки разделителя, 485
определение позиции и длины совпадения, 199
относительные пути в Windows, 590, 592
отрицание внутри символьных классов, 57
отрицательная опережающая проверка, 120
отрицательная ретроспективная проверка, 120

П

параметры, в файлах INI, 611
параметры регулярных выражений, установка, 171
пароли, проверка сложности, 377
  запрет появления трех и более идентичных символов, следующих подряд, 379
  любые алфавитные символы Юникода в верхнем регистре, 378
  любые алфавитные символы Юникода в нижнем регистре, 378
  один или более алфавитных символов в верхнем регистре, 378
  один или более алфавитных символов в нижнем регистре, 378
  один или более специальных символов, 379
  одна или более цифр, 379
  пример решения на JavaScript
    простое решение, 379
    с определением оценки надежности пароля, 381
    с подсчетом числа удовлетворяемых требований, 380
  проверка нескольких требований в одном регулярном выражении, 385
  только видимые символы ASCII и пробелы, 378
парсер, 284
парсинг текста, 234, 284
первое вхождение строки, сохранение, 434, 437
переменное число повторений, 104
пересечение символьных классов, 61
повторение при сопоставлении
  минимальное и максимальное число повторений, 107
  повторяющиеся части регулярного выражения, 103
  предотвращение бесконтрольных повторений, 114
повторное использование частей совпадений, 245
повторяющиеся пробельные символы, удаление, 447
повторяющиеся слова, поиск, 429
позитивная опережающая проверка
  поиск слов на любом расстоянии друг от друга, 428
позиция совпадения, определение, 199
поиск, 441
  адресов URL в тексте, 529
    заключенных в кавычки, 532
    заключенных в скобки, 533
  адресов, содержащих номер почтового ящика, 371
  определенных атрибутов в тегах XML, 653
  построчный, 280
  слов, 400
    близко расположенных, 421
    в XML-подобных комментариях, 667
    за которыми не следует указанное слово, 413
    из списка, 403
    которым не предшествует определенное слово, 415
    любых, за исключением некоторых, 411
    повторяющихся, 429
    похожих, 406
строк
  не содержащих определенное слово, 441
  повторяющихся, 433
  совпадение со всей строкой, 438, 441
  содержащих определенное слово, 438
тегов XML, 612
чисел
  восьмеричных, 463
  двоичных, 462
  десятичных, 464
  целых, 454
  шестнадцатеричных, 458

поиск с заменой
вставка текстового литерала в замещающий текст, 133
замена всех совпадений, 237
замена совпадений фрагментами, сгенерированными в программном коде, 251
замена специальных символов HTML мнемоническими ссылками, 647
замещение всех совпадений внутри совпадений с другим регулярным выражением, 258
замещение всех совпадений между совпадениями с другим регулярным выражением, 260
повторное использование частей совпадений, 245
с помощью регулярных выражений, 22
тегов в языках разметки, 631
положительная опережающая проверка, 119
положительная ретроспективная проверка, 118
поля, в файлах CSV, 610
извлечение, 677
порты, извлечение из адреса URL, 553
последнее вхождение строки, сохранение, 434, 436
построчный поиск, 280
похожие слова, поиск, 406
почтовые индексы
Великобритании, проверка, 369
Канады, проверка, 368
США, проверка, 367
правый контекст, 143
предотвращение бесконтрольных повторений, 114
предотвращение добавления запятых после десятичной точки, 488
преобразование имен из одного формата в другой, 373
пробельные символы
в режиме свободного форматирования, 132
категория Юникода, 78
повторяющиеся, удаление, 447
сопоставление в символьных классах, 60
удаление ведущих и завершающих пробельных символов в строках, 442
удаление из европейских регистрационных номеров плательщиков НДС, 394, 396
удаление из номера кредитной карты, 387, 390
проверка, 309
адресов URL, 525, 540
адресов электронной почты, 301
времени в традиционных форматах, 327
дат в традиционных форматах, 316
дат и времени в формате ISO 8601, 330
доменных имен, 562
европейских регистрационных номеров плательщиков НДС, 393
комментариев XML, 663
путей в Windows, 585
строк URN, 537
телефонных номеров, 308
международных, 314
проверка ввода
номеров ISBN, 358
номеров кредитных карт, 387, 392
номеров социального страхования, 356
ограничение возможности ввода алфавитно-цифровыми символами, 338
ограничение возможности ввода символами ANSI, 340
ограничение длины строки текста, 342
ограничение числа строк в тексте, 348
почтовых индексов, 367
сложность пароля, 377
утвердительных ответов, 354
проверка возможности совпадения в пределах испытуемой строки, 179
проверка полученных совпадений в программном коде, 227
проверка с нулевой длиной совпадения, 119
проверка совпадения со всей испытуемой строкой, 186
проверка соседних символов, 118, 344, 402
отрицательная, 120
проверка условия, 130
проверка условия, при сопоставлении, 127
простой текст, преобразование в HTML, 646
пустые обратные ссылки, 425
пути в файловой системе (Windows)
выделение элементов, 589
извлечение буквы устройства, 594
извлечение имени сервера, 595
извлечение имен папок, 596
извлечение имен файлов, 599
извлечение расширений файлов, 600
проверка, 585
удаление недопустимых символов, 601
пути, извлечение из адреса URL, 556

Р

разбиение строки, 266
  с сохранением совпадений, 275
разделители в файлах CSV, изменение, 672
разделители строк, 352
разделы, в файлах INI, 611
разность символьных классов, 60, 61
разрывы строк в документах HTML, 647
расширения (файлов), извлечение из путей в Windows, 600
расширяемый протокол представления информации (Extensible Provisioning Protocol, EPP), 316
расширяемый язык разметки (Extensible Markup Language, XML), 608
  декодирование мнемоник, 650
  добавление атрибутов в теги <table>, 658
  поиск определенных атрибутов, 653
  поиск слов в комментариях, 667
  поиск тегов, 612, 615
  сопоставление с именами, 639
  удаление всех тегов, за исключением выбранных, 635
  удаление комментариев, 662
расширяемый язык разметки гипертекста (Extensible Hypertext Markup Language, XHTML), 607
  добавление атрибутов в теги <table>, 658
  поиск тегов, 612
  удаление всех тегов, за исключением выбранных, 635
регистр символов
  нечувствительность к регистру в символьных классах, 60
  нечувствительность к регистру символов, 304
  поиск без учета регистра символов, 52
регулярные выражения
  в стиле языка Perl, 18
  литералы, извлечение из исходного кода, 511
  экранирование метасимволов, 448
режим свободного форматирования, 131
  в Python, 159
ретроспективная проверка, 118, 344, 402
  бесконечное и ограниченное повторение, 513
  имитация ретроспективной проверки, 416, 418
  имитация с применением сохраняющей группировки, 126
  негативная, 415
  отрицательная, 120
  положительная, 118
римские числа, 490

С

свободное форматирование, режим, 131
  установка, 172
«символам ^ и $ соответствуют границы строк», режим, 171
символ торговой марки, сопоставление, 74
символы ANSI, ограничение ввода, 340
символы ISO-8859-1, ограничение ввода, 340
символы Windows-1252, ограничение ввода, 340
символы, категории Юникода, 76
символы слова, 59, 73
символьные классы, 57, 310
  в режиме свободного форматирования, 131
  метасимволы Юникода, 87
  нечувствительность к регистру символов, 60
  объединение и пересечение, 61
  разность, 60
  сокращения, 58
слова
  поиск близко расположенных слов, 421
  поиск в XML-подобных комментариях, 667
  поиск любого слова, за которым не следует указанное слово, 413
  поиск любого слова, которому не предшествует определенное слово, 415
  поиск любых слов, за исключением некоторых, 411
  поиск одного слова из списка, 403
  поиск определенного слова, 400
  поиск повторяющихся слов, 429
  поиск похожих слов, 406
  поиск строк, не содержащих определенное слово, 441
  поиск строк, содержащих определенное слово, 438
  сопоставление с целыми словами, 71
  экранирование метасимволов, 448
сложность пароля, 377
  запрет появления трех и более идентичных символов, следующих подряд, 379
  любые алфавитные символы Юникода в верхнем регистре, 378

любые алфавитные символы Юникода
в нижнем регистре, 378
один или более алфавитных символов
в верхнем регистре, 378
один или более алфавитных символов
в нижнем регистре, 378
один или более специальных символов,
379
одна или более цифр, 379
пример решения на JavaScript
простое решение, 379
с определением оценки надежности
пароля, 381
с подсчетом числа удовлетворяемых
требований, 380
проверка нескольких требований
в одном регулярном выражении, 385
только видимые символы ASCII
и пробелы, 378
смешанная форма записи (IPv6), 570, 576,
580
компактная, 574, 576, 583, 584
совпадения внутри другого совпадения,
231
совпадения нулевой длины, 69
создание объектов регулярных выраже-
ний, 162
сокращенные формы записи символьных
классов, 58
сопоставление, 441
без учета регистра символов, 52, 304
в начале и/или в конце строки, 65
внутри совпадений с другим регуляр-
ным выражением, 258
вставка контекста совпадения
в замещающий текст, 142
замена совпадений с повторным исполь-
зованием частей совпадений, 245
обход всех совпадений в цикле, 219
поиск совпадения внутри другого
совпадения, 231
проверка условия, 127
с адресами IPv4, 565
с адресами IPv6, 569
с заголовком раздела в файле INI, 681
с именами XML, 639
с литеральным текстом, 50
с любым символом, 62
с одной из нескольких альтернатив, 90
с одним символом из нескольких, 56
с парами имя-значение в файле INI, 684
с повторением
минимальное и максимальное
число повторений, 107
повторяющиеся части регулярного
выражения, 103
предотвращение бесконтрольных
повторений, 114
с разделом в файле INI, 683
с ранее совпавшим текстом, 96
с тегами в языках разметки, 612
с целыми словами, 71
сортировка строк, 433, 435
сохраняющая группировка, 92, 499
извлечение информации о совпадении,
205
именованное сохранение, 99, 140, 211,
248
группы с одинаковыми именами
(.NET), 326
имитация ретроспективной проверки,
126
использование пустых обратных
ссылок, 425
повторное использование частей
совпадений, 245
повторный поиск соответствия с ранее
совпавшим текстом, 96
поиск повторяющихся слов, 429
проверка условия, 127
ссылки на несуществующие группы,
140
ссылки, недействительные, сообщения
в файле журнала веб-сервера, 521
стандартная форма записи (IPv6), 569
строка из определенного пространства
имен (Namespace Specific String, NSS),
539
строки
в апострофах (PHP), 157
в кавычках
C#, 155
Java, 155
Perl, 157
PHP, 156
VB.NET, 155
в тройных кавычках (Python), 158
извлечение из исходного кода, 505, 509
проверка адресов URL, 525
проверка длины ввода, 342
проверка строк URN, 537
разбиение, 266
экранирование метасимволов, 448
строки запросов, извлечение из адреса
URL, 560
строки текста, 433
не содержащие определенное слово, 441
содержащие определенное слово, 438

сортировка с целью удаления дубликатов, 433, 435
удаление ведущих и завершающих пробельных символов, 442
удаление дубликатов, 433
удаление повторяющихся пробельных символов, 447
схемы, извлечение из адреса URL, 546
сырые строки (Python), 158

Т

теги, языки разметки, 605
добавление атрибутов в теги <table>, 658
замещение одного тега другим, 631
сопоставление, 612
удаление всех тегов, за исключением выбранных, 635
телефонные номера
международные, 314
проверка, 308
точка – это все (dot all), режим, 63
точке соответствуют границы строк, режим, 63, 171

У

удаление
ведущих нулей, 465
недопустимых символов из имен файлов, 601
повторяющихся пробельных символов, 447
универсальные адреса URL, проверка, 540
унифицированное имя ресурса (Uniform Resource Name, URN), проверка, 537
управление возвратами, 623
управляющие символы
категория Юникода, 79
установка параметров регулярных выражений, 171
утвердительные ответы, 354

Ф

файлы инициализации (INI), 611
фиксированная длина совпадения
с ретроспективной проверкой, 417
фиксированное число повторений, 104
формат даты и времени ISO 8601, 330
форматирование
имен, 373
проверка времени
в традиционных форматах, 327
в формате ISO 8601, 330
проверка дат, 316
в формате ISO 8601, 330
телефонных номеров, 308
форматы дат, проверка, 316
фрагмент, извлечение из адреса URL, 561

Ц

целые числа
в определенном диапазоне, поиск, 467, 474
добавление разделителей групп разрядов, 484
поиск и идентификация, 454
с разделителями групп разрядов, 477, 483
удаление ведущих нулей, 465
цифры (числа), 456
категория Юникода, 79
соответствия в символьных классах, 58

Ч

числа
IPv4, адреса, сопоставление, 565
IPv6, адреса, сопоставление, 569
вещественные, 479
восьмеричные, поиск, 463
двоичные, поиск, 462
десятичные, поиск, 464
добавление разделителей групп разрядов, 484
европейские регистрационные номера плательщиков НДС, проверка, 393
номера ISBN, проверка, 358
номера кредитных карт, проверка, 387
проверка номеров социального страхования, 356
римские, 490
с разделителями групп разрядов, 483
удаление ведущих нулей, 465
целые
в определенном диапазоне, поиск, 467
поиск и идентификация, 454
с разделителями групп разрядов, 477
шестнадцатеричные
в определенном диапазоне, поиск, 474
поиск, 458
числовые константы, извлечение из исходного кода, 499
числовые ссылки на символы, 606

Ш

шестнадцатеричные числа, поиск, 458
    в определенном диапазоне, 474

Э

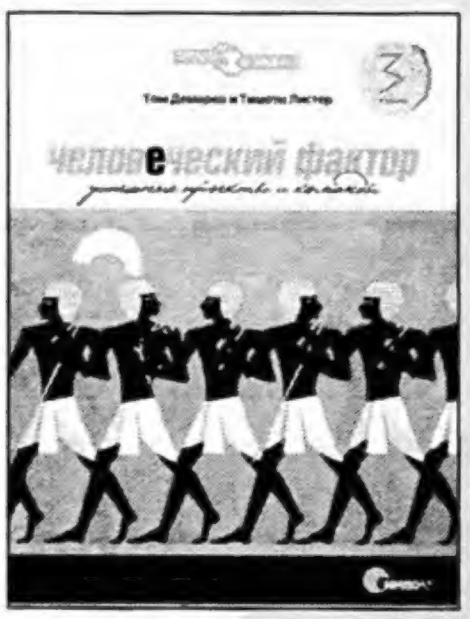
экранирование, 51
    блока, 51, 453
    внутри символьных классов, 57
    как и когда, 449
    символы в замещающем тексте, 134
электронная почта, проверка адресов, 301
элементы, языки разметки, 605

Ю

Юникод, 74

Я

язык разметки гипертекста (Hypertext Markup Language, HTML), 604
    добавление атрибутов в теги <table>, 658
    добавление тегов <p> в простой текст, 646
    замещение тега <b> тегом , 631
    поиск тегов, 612
    удаление всех тегов, за исключением выбранных, 635
язык разметки гипертекста, расширяемый (Extensible Hypertext Markup Language, XHTML), 607
язык разметки, расширяемый (Extensible Markup Language, XML), 608
языки, алфавиты Юникода, 84
языки программирования
    ActionScript, 148
    C, 148
    C++, 148
    C#, 145
    Delphi, 149
    Delphi Prism, 150
    ECMAScript, 146
    Groovy, 150
    Java, 145
    JavaScript, 146
    Perl, 147
    PHP, 146
    Python, 147
    Ruby, 147
    Scala, 151
    VB.NET, 145, 151
якори, 66



Том ДЕМАРКО и Тимоти ЛИСТЕР

Человеческий фактор: успешные проекты и команды, 3-е издание

288 стр., книга в продаже

Немногие книги о компьютерах оказали такое заметное влияние на управление разработкой программного обеспечения, как «Человеческий фактор». Уникальное озарение этой книги, долгие годы сохранявшей свое положение в списке бестселлеров: самые сложные проблемы разработки программного обеспечения носят не технический, но социальный характер. Эти человеческие проблемы решать не просто, однако, решив их, вы увеличите шансы на успех до верхнего предела.

В третьем издании появилось шесть новых глав и внесены многочисленные правки в основной текст, так что теперь книга лучше соответствует современным средам разработки и современным проблемам. В частности, книга теперь освещает патологии лидерства, которые раньше за патологии не считались, эволюционирующую культуру собраний, а еще растущее понимание того, что некоторые наши инструменты служат скорее якорями, а не двигателями.

Каждый, кому необходимо управлять проектом по разработке ПО или целой организацией такого рода, найдет на страницах этой книги множество ценных советов.

Ричард СТИВЕНС, Стивен РАГО

UNIX. Профессиональное программирование, 3-е издание

1104 стр., книга в продаже

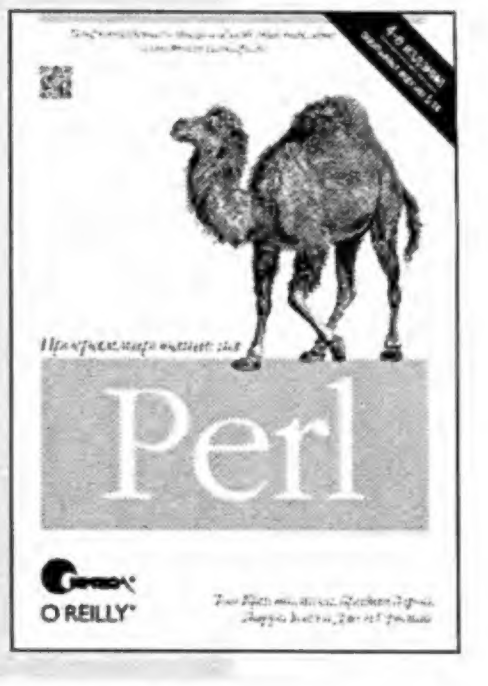
Издание представляет собой подробнейшее справочное руководство для любого профессионального программиста, работающего с UNIX. Стив Раго, коллега Рича Стивенса, вновь полностью обновил классический труд, сохранив точность и стиль оригинала. Новое, третье, издание охватывает современные ведущие платформы, отражает новейшие технические достижения и передовую практику и соответствует 4-й версии Single UNIX Specification.

Помимо основ (файлы, каталоги и процессы) рассматриваются более сложные темы, такие как обработка сигналов и терминальный ввод/вывод, многопоточная модель выполнения и межпроцессное взаимодействие с применением сокетов. Третье издание охватывает более 70 новых интерфейсов, включая функции POSIX асинхронного ввода/вывода, циклические блокировки, барьеры и семафоры POSIX. Примеры протестированы на 4-х наиболее популярных платформах: Solaris 10, Mac OS X 10.6.8 (Darwin 10.8.0), FreeBSD 8.0 и Ubuntu 12.04 (основана на ядре Linux 3.2). Описания более 400 системных вызовов и функций сопровождаются короткими примерами законченных программ, которые наглядно демонстрируют порядок их применения, входные аргументы и возвращаемые значения.

Том КРИСТИАНСЕН, брайан д фой,
Ларри УОЛЛ и Джон ОРВАНТ

Программирование на Perl, 4-е издание

1048 стр., книга в продаже



Четвертое издание этой книги ждали в России и программисты, и системные администраторы. И вот обновление описания языка Perl, продолжавшего активно развиваться в течение последних пяти лет. На этот раз в «Кэмэл» обсуждается текущая версия Perl 5.14 и дается обзор некоторых особенностей готовящейся к выходу версии Perl 5.16.

Все большую значимость в обработке текстов приобретает Юникод, а Perl предлагает лучшую и самую безболезненную поддержку этого стандарта, тесно интегрируя Юникод во все сферы, в том числе в такой популярный механизм языка Perl, как регулярные выражения. И Юникоду, и регулярным выражениям уделено много внимания в книге. Четвертое издание охватывает такие важные особенности языка Perl, как новые ключевые слова и синтаксические конструкции, уровни ввода/вывода и кодировки, новые escape-последовательности, поддержка стандарта Unicode 6.0, групповые графемы и свойства символов Юникода, именованные сохраняющие группы в регулярных выражениях, рекурсивные и грамматические шаблоны, современные передовые приемы программирования. Справочник по языку Perl перед вами!

Джеффри ФРИДЛ

Регулярные выражения, 3-е издание

608 стр., книга в продаже

Эта книга откроет перед вами секрет высокой производительности. Тщательно продуманные регулярные выражения помогут избежать долгих часов утомительной работы и решить свои проблемы за 15 секунд. Ставшие стандартной возможностью во многих языках программирования и популярных программных продуктах, включая Perl, PHP, Java, Python, Ruby, MySQL, VB.NET, C# (и другие языки платформы .NET), регулярные выражения позволят вам автоматизировать сложную и тонкую обработку текста.

Написанное простым и доступным языком, это издание позволит программистам легко разобраться в столь сложной теме. Рассматривается принцип действия механизма регулярных выражений, сравниваются функциональные возможности различных языков программирования и инструментальных средств, подробно обсуждается оптимизация, которая дает основную экономию времени! Вы научитесь правильно конструировать регулярные выражения для самых разных ситуаций, а большое число сложных примеров даст возможность сразу же использовать предлагаемые ответы для выработки элегантных и экономичных практических решений широкого круга проблем.

Издательство "СИМВОЛ-ПЛЮС"

Основано в 1995 году

О нас

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.

Где купить

Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

Санкт-Петербург:

главный офис издательства –
В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 380-5007

Москва:

московский филиал издательства –
ул. 2-я Магистральная, д. 14В
(м. Полежаевская/Беговая),
тел. (495) 638-5305

Заказ книг

через Интернет http://www.symbol.ru

Бесплатный каталог книг высылается по запросу.

Приглашаем к сотрудничеству

www.symbol.ru

Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!

O'REILLY®

Регулярные выражения. Сборник рецептов

Отбросьте все свои догадки о регулярных выражениях. Эта книга содержит все необходимое для решения широкого спектра практических задач в виде более 140 проверенных рецептов. С ее помощью начинающие смогут обрести базовые навыки и освоить основные инструменты, а программисты и опытные пользователи найдут в ней множество важных мелочей. Каждый рецепт содержит шаблоны, пригодные для немедленного использования.

Второе обновленное издание охватывает диалекты регулярных выражений, используемые в языках программирования C#, Java, JavaScript, Perl, PHP, Python, Ruby и VB.NET. Вы познакомитесь с новыми приемами, научитесь избегать ловушек в разных диалектах и благодаря огромной библиотеке готовых решений сможете сэкономить свое время..

С помощью этой книги вы:

- Усвоите основы регулярных выражений благодаря подробному учебному руководству
- Научитесь использовать регулярные выражения в любом из охватываемых языков программирования
- Познакомитесь с отличиями диалектов регулярных выражений в разных языках программирования
- Узнаете, как проверять и форматировать ввод пользователя
- Овладеете приемами поиска и обработки слов, специальных символов и строк текста
- Научитесь выявлять целые и вещественные числа в различных форматах представления
- Узнаете приемы парсинга исходного программного кода и файлов журналов
- Найдете решения по использованию регулярных выражений для работы с адресами URL, путями и IP-адресами
- Овладеете приемами работы с разметкой HTML, XML и с форматами обмена данными
- Познакомитесь с малоизвестными особенностями регулярных выражений и приемами их использования

«Регулярные выражения – неустаревающая технология, и книга «Регулярные выражения. Сборник рецептов» еще долгие годы будет оставаться надежным источником информации и вдохновения.»

Бен Найдел (Ben Nadel), главный инженер-программист в Epicenter Consulting

КАТЕГОРИЯ: ПРОГРАММИРОВАНИЕ
УРОВЕНЬ ПОДГОТОВКИ ЧИТАТЕЛЕЙ: СРЕДНИЙ

Издательство «Символ-Плюс»
(812) 380-5007, (495) 638-5305

ISBN 978-5-93286-221-6

9 785932 862216

СИМВОЛ®
www.symbol.ru

O'REILLY®